```
In [1]:
%matplotlib inline
%load_ext autoreload
%autoreload 2

import matplotlib.pyplot as plt

import numpy as np
from skimage.transform import resize
from sklearn.naive_bayes import GaussianNB, BernoulliNB
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
import os

from aml_utils import show_test_cases, test_case_checker, perform_computation
```

Libraries such as `math` are neither as accurate nor as efficient as `numpy`.

Note: Do not import or use any other libraries other than what is already imported above.

*Assignment Summary

You may find it useful to read Chapter 1 and 2 of the textbook about classification.

The MNIST dataset is a dataset of 60,000 training and 10,000 test examples of handwritten digits, originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. It is very widely used to check simple methods. There are 10 classes in total ("0" to "9"). This dataset has been extensively studied, and there is a history of methods and feature constructions at https://en.wikipedia.org/wiki/MNIST_database and at the original site, http://yann.lecun.com/exdb/mnist/ . You should notice that the best methods perform extremely well.

There is also a version of the data that was used for a Kaggle competition. I used it for convenience so I wouldn't have to decompress Lecun's original format. I found it at http://www.kaggle.com/c/digit-recognizer .

Regardless of which format you find the dataset stored in, the dataset consists of 28 x 28 images. These were originally binary images, but appear to be grey level images as a result of some anti-aliasing. We will ignore mid grey pixels (there aren't many of them) and call dark pixels "ink pixels", and light pixels "paper pixels"; you can modify the data values with a threshold to specify the distinction, as described here https://en.wikipedia.org/wiki/Thresholding_(image_processing) . The digit has been centered in the image by centering the center of gravity of the image pixels, but as mentioned on the original site, this is probably not ideal. Here are some options for re-centering the digits that I will refer to in the exercises.

- **Untouched**: Do not re-center the digits, but use the images as is.
- **Bounding box**: Construct a 20 x 20 bounding box so that the horizontal (resp. vertical) range of ink pixels is centered in the box.
- **Stretched bounding box**: Construct a 20 x 20 bounding box so that the horizontal (resp. vertical) range of ink pixels runs the full horizontal (resp. vertical) range of the box. Obtaining this representation will involve rescaling image pixels: you find the horizontal and vertical ink range, cut that out of the original image, then resize the result to 20 x 20. Once the image has been re-centered, you can compute features.

Here are some pictures, which may help.

alt text

Part 2A Investigate classifying MNIST using naive Bayes. Fill in the accuracy values for the four combinations of Gaussian v. Bernoulli distributions and untouched images v. stretched bounding boxes in a table like this. Please use 20 x 20 for your bounding box dimensions.

Accuracy	Gaussian	Bernoulli
Untouched images		
Stretched bounding box		

Which distribution (Gaussian or Bernoulli) is better for untouched pixels? Which is better for stretched bounding box images? **Part 2B** Investigate classifying MNIST using a decision forest. For this you should use a library. For your forest construction, try out and compare the combinations of parameters shown in the table (i.e. depth of tree, number of trees, etc.) by listing the accuracy for each of the following cases: untouched raw pixels; stretched bounding box. Please use 20 x 20 for your bounding box dimensions. In each case, fill in a table like those shown below.

Accuracy	depth = 4	depth = 8	depth = 16
#trees = 10			
#trees = 20			
#trees = 30			

0. Data

Since the MNIST data (http://yann.lecun.com/exdb/mnist/) is stored in a binary format, we would rather have an API handle the loading for us.

Pytorch (https://pytorch.org/) is an Automatic Differentiation library that we may see and use later in the course.

Torchvision (https://pytorch.org/docs/stable/torchvision/index.html?highlight=torchvision#module-torchvision) is an extension library for pytorch that can load many of the famous data sets painlessly.

We already used Torchvision for downloading the MNIST data. It is stored in a numpy array file that we will load easily.

```
In [2]:
if os.path.exists('../ClassifyingImages-lib/mnist.npz'):
    npzfile = np.load('../ClassifyingImages-lib/mnist.npz')
    train_images_raw = npzfile['train_images_raw']
    train_labels = npzfile['train_labels']
    eval_images_raw = npzfile['eval_images_raw']
    eval_labels = npzfile['eval_labels']
else:
    import torchvision
    download_ = not os.path.exists('../ClassifyingImages-lib/mnist.npz')
    data_train = torchvision.datasets.MNIST('mnist', train=True, transform=None, targe
    data_eval = torchvision.datasets.MNIST('mnist', train=False, transform=None, targe

    train_images_raw = data_train.data.numpy()
    train_labels = data_train.targets.numpy()
    eval_images_raw = data_eval.data.numpy()
    eval_labels = data_eval.targets.numpy()

    np.savez('../ClassifyingImages-lib/mnist.npz', train_images_raw=train_images_raw,
             eval_images_raw=eval_images_raw, eval_labels=eval_labels)
```

0.1 Getting to Know The Imported Data

Let's visibly check and see what we have imported.

1. What is the order of dimensions? (We need to check whether the number of samples is the first dimension or the last dimension)
2. What is the training data type?
3. What is the labels shape and data type? Is it one-hot encoded?
4. What are the pixel value ranges? Is it between 0 and 1? Or is it between 0 and 255? or something else?
5. How is an ink (resp. background) pixel represented? Is it represented with 0? Is it represented with 1? Is it represented with 255? or something else?

The following cells should help you answer these questions in order.

1. What is the order of dimensions? (We need to check whether the number of samples is the first dimension or the last dimension)

```
In [3]: train_images_raw.shape
Out[3]: (60000, 28, 28)
```

A: 3rd order, 3D. 1st dim is 60k obs.

1. What is the training data type?

```
In [4]: train_images_raw.dtype
Out[4]: dtype('uint8')
```

A: uint8

1. What is the labels shape and data type? Is it one-hot encoded?

```
In [5]: train_labels.shape, train_labels.dtype
Out[5]: ((60000,), dtype('int64'))
```

Labels are vector of 60k int64.

1. What are the pixel value ranges? Is it between 0 and 1? Or is it between 0 and 255? or something else?

```
In [6]: train_labels[:10]
Out[6]: array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4])
```

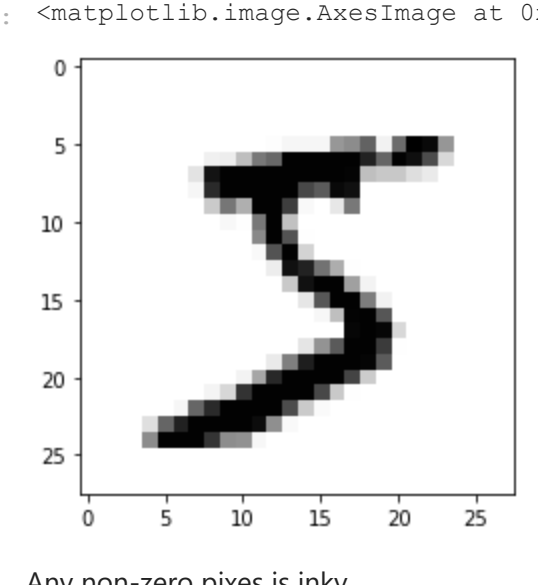
```
In [7]: train_images_raw[0].min(), train_images_raw[0].max()
Out[7]: (0, 255)
```

(0, 255)

1. How is an ink (resp. background) pixel represented? Is it represented with 0? Is it represented with 1? Is it represented with 255? or something else?

```
In [8]:
for row_im in train_images_raw[0]:
    print(row_im.tolist())
plt.imshow(train_images_raw[0], cmap='Greys')
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 18, 18, 18, 126, 136, 175, 26, 166, 255, 247,
127, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 30, 36, 94, 154, 170, 253, 253, 253, 253, 253, 225, 172, 253,
242, 195, 64, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 49, 238, 253, 253, 253, 253, 253, 253, 253, 253, 251, 93, 82, 8
2, 56, 39, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 18, 219, 253, 253, 253, 253, 253, 198, 182, 247, 241, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 80, 156, 107, 253, 253, 205, 11, 0, 43, 154, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 14, 1, 154, 253, 90, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 139, 253, 190, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 190, 253, 70, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 35, 241, 225, 160, 108, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 81, 240, 253, 253, 119, 25, 0, 0, 0, 0, 0, 0,
0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 45, 186, 253, 253, 150, 27, 0, 0, 0, 0, 0,
0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 93, 252, 253, 187, 0, 0, 0, 0, 0, 0,
0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 249, 253, 249, 64, 0, 0, 0, 0, 0,
0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 46, 130, 183, 253, 253, 207, 2, 0, 0, 0, 0,
0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 39, 148, 229, 253, 253, 253, 250, 182, 0, 0, 0,
0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 24, 114, 221, 253, 253, 253, 253, 201, 78, 0, 0, 0, 0,
0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 23, 66, 213, 253, 253, 253, 253, 198, 81, 2, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 18, 171, 219, 253, 253, 253, 253, 195, 80, 9, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
[0, 0, 0, 0, 55, 172, 226, 253, 253, 253, 253, 244, 133, 11, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 136, 253, 253, 253, 212, 135, 132, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Out[8]: <matplotlib.image.AxesImage at 0x7fb5ec9df6d0>
```



Any non-zero pixes is inky.

0.2 Thresholding

Task 1

Write the function `get_thresholded` that does image thresholding and takes following the arguments:

1. `images_raw` : A numpy array. Do not assume anything about its shape, dtype or range of values. Your function should be careless about these attributes.
2. `threshold` : A scalar value.

and returns the following:

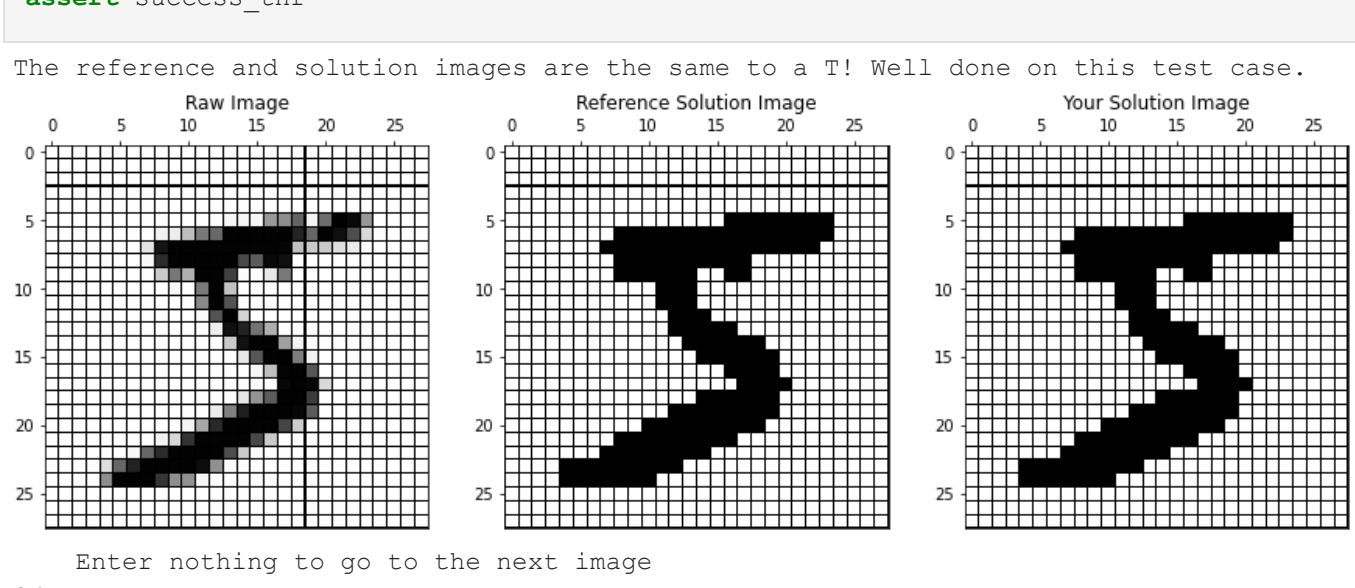
- `threshed_image` : A numpy array with the same shape as `images_raw`, and the `bool` dtype. This array should indicate whether each elemelent of `images_raw` is **greater than or equal to** `threshold`.

```
In [9]:
def get_thresholded(images_raw, threshold):
    """
    Perform image thresholding.

        Parameters:
                images_raw (np,array): Do not assume anything about its shape, dtype o
                Your function should be careless about these attributes.
                threshold (int): A scalar value.

        Returns:
                threshed_image (np.array): A numpy array with the same shape as images
                This array should indicate whether each elemelent of images_raw is gre
                threshold.
    """
    threshed_image = images_raw >= threshold

    return threshed_image
```

```
In [10]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [11]:
(orig_image, ref_image, test_im, success_thr) = show_test_cases(lambda x: get_thresho

assert success_thr
```

```
The reference and solution images are the same to a T! Well done on this test case.
```




```
    Enter nothing to go to the next image
or
    Enter "s" when you are done to recieve the three images.
        **Don't forget to do this before continuing to the next step.**
s
```

Warning:

Do not leave the previous cell hanging; unless you enter "s" to stop it, you cannot evaluate other cells.

```
In [12]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [13]:
# Checking against the pre-computed test database
test_results = test_case_checker(get_thresholded, task_id=1)
assert test_results['passed'], test_results['message']
```

```
In [14]:
train_images_threshed = get_thresholded(train_images_raw, threshold=20)
eval_images_threshed = get_thresholded(eval_images_raw, threshold=20)
```

0.3 Creating "Bounding Box" Images

0.3.1 Finding Inky Rows

Task 2

Write the function `get_is_row_inky` that finds the rows with ink pixels and takes following the arguments:

- `images` : A numpy array with the shape `(N,height,width)`, where
  - `N` is the number of samples and could be anything,
  - `height` is each individual image's height in pixels (i.e., number of rows in each image),
  - and `width` is each individual image's width in pixels (i.e., number of columns in each image).
    - Do not assume anything about `images`'s dtype or the number of samples or the `height` or the `width`.

and returns the following:

- `is_row_inky` : A numpy array with the shape `(N, height)`, and the `bool` dtype.
  - `is_row_inky[i,j]` should be True if **any** of the pixels in the `j` th row of the `i` th image was an ink pixel, and False otherwise.

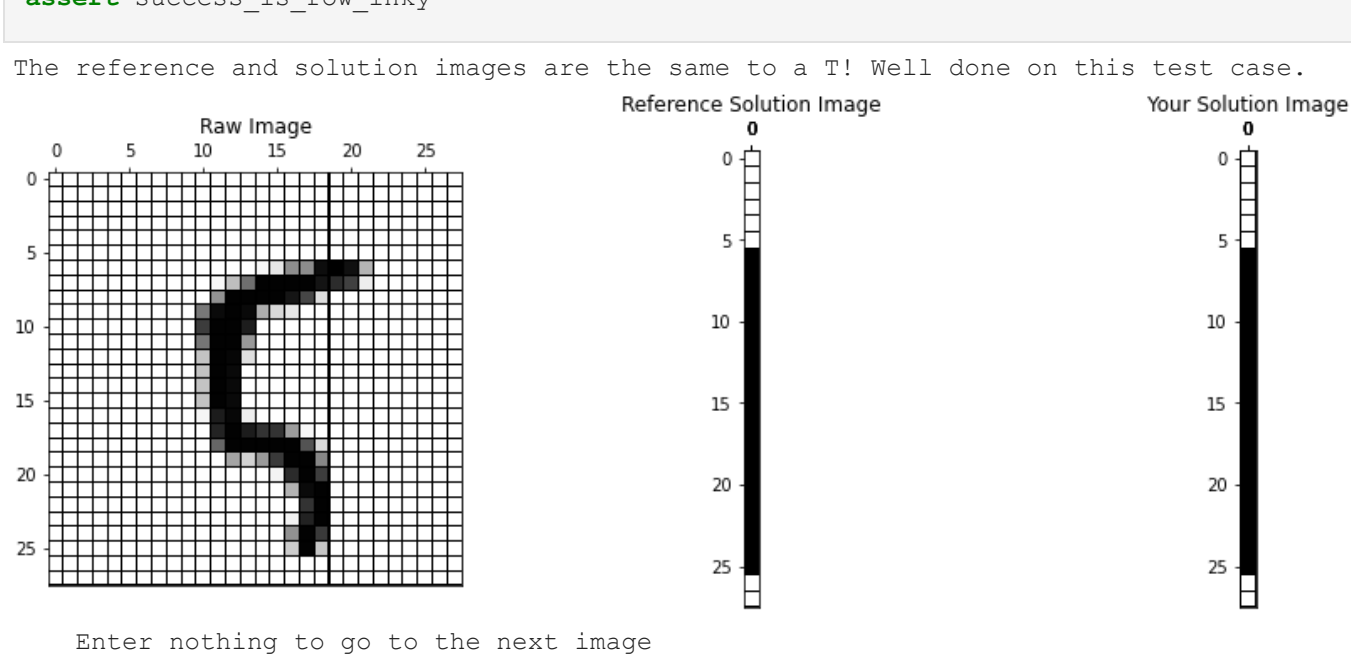
```
In [15]:
#train_images_raw.shape #= (60000, 28, 28)
#np.sum(train_images_raw, axis = 1).shape(60000, 28)
(np.sum(train_images_raw, axis = 1) > 0).shape
Out[15]: (60000, 28)
```

```
In [16]:
def get_is_row_inky(images):
    """
    Finds the rows with ink pixels.

        Parameters:
                images (np.array): A numpy array with the shape (N, height, width)

        Returns:
                is_row_inky (np.array): A numpy array with the shape (N, height), and
    """
    is_row_inky = np.sum(images, axis = 2) > 0

    return is_row_inky
```

```
In [17]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [18]:
(orig_image, ref_image, test_im, success_is_row_inky) = show_test_cases(lambda x: np.
                                                                       task_id='2_V')

assert success_is_row_inky
```

```
The reference and solution images are the same to a T! Well done on this test case.
```




```
    Enter nothing to go to the next image
or
    Enter "s" when you are done to recieve the three images.
        **Don't forget to do this before continuing to the next step.**
s
```

Warning:

Do not leave the previous cell hanging; unless you enter "s" to stop it, you cannot evaluate other cells.

```
In [19]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [20]:
# Checking against the pre-computed test database
test_results = test_case_checker(get_is_row_inky, task_id=2)
assert test_results['passed'], test_results['message']
```

0.3.2 Finding Inky Columns

Task 3

Similar to `get_is_row_inky`, Write the function `get_is_col_inky` that finds the columns with ink pixels and takes following the arguments:

- `images` : A numpy array with the shape `(N,height,width)`, where
  - `N` is the number of samples and could be anything,
  - `height` is each individual image's height in pixels (i.e., number of rows in each image),
  - and `width` is each individual image's width in pixels (i.e., number of columns in each image).
    - **Note**: Do not assume anything about `images`'s dtype or the number of samples or the `height` or the `width`.

and returns the following:

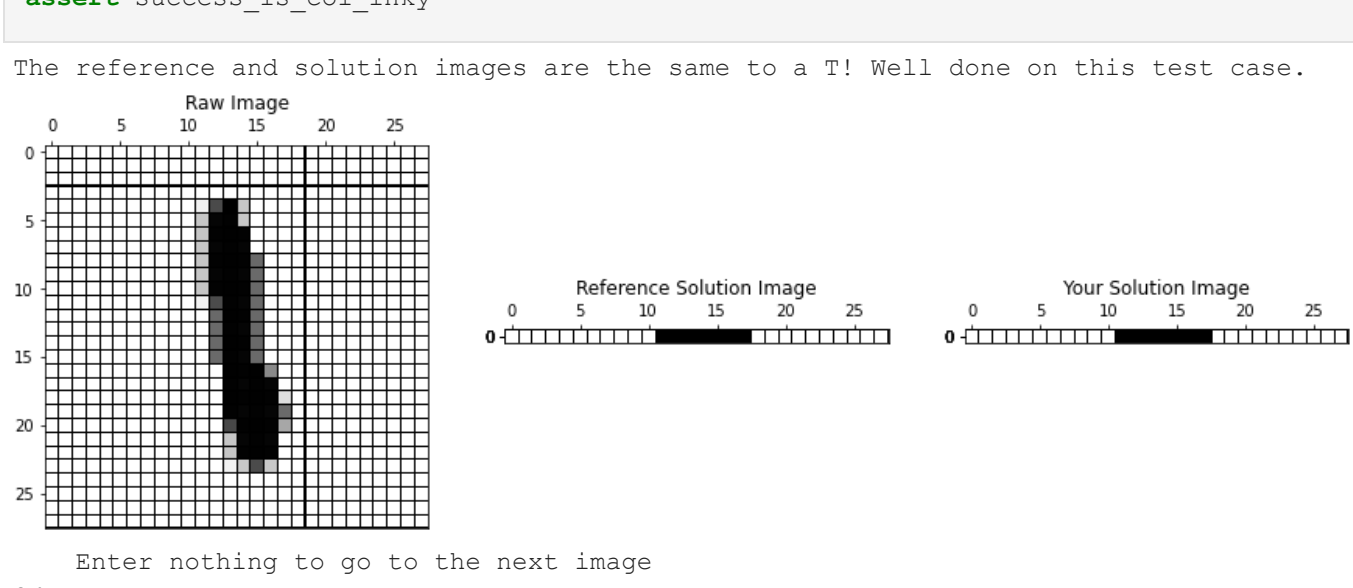
- `is_col_inky` : A numpy array with the shape `(N, width)`, and the `bool` dtype.
  - `is_col_inky[i,j]` should be True if **any** of the pixels in the `j` th column of the `i` th image was an ink pixel, and False otherwise.

```
In [21]:
def get_is_col_inky(images):
    """
    Finds the columns with ink pixels.

        Parameters:
                images (np.array): A numpy array with the shape (N,height,width).

        Returns:
                is_col_inky (np.array): A numpy array with the shape (N, width), and t
    """

    is_col_inky = np.sum(images, axis = 1) > 0

    return is_col_inky
```

```
In [22]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [23]:
(orig_image, ref_image, test_im, success_is_col_inky) = show_test_cases(lambda x: np.
                                                                       task_id='3_V')

assert success_is_col_inky
```

```
The reference and solution images are the same to a T! Well done on this test case.
```




```
    Enter nothing to go to the next image
or
    Enter "s" when you are done to recieve the three images.
        **Don't forget to do this before continuing to the next step.**
s
```

Warning:

Do not leave the previous cell hanging; unless you enter "s" to stop it, you cannot evaluate other cells.

```
In [24]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [25]:
# Checking against the pre-computed test database
test_results = test_case_checker(get_is_col_inky, task_id=3)
assert test_results['passed'], test_results['message']
```

0.3.3.1 Getting the First Inky Rows

Task 4

Write the function `get_first_ink_row_index` that finds the first row containing ink pixels and takes following the arguments:

- `is_row_inky` : A numpy array with the shape `(N, height)`, and the `bool` dtype. This is the output of the `get_is_row_inky` function that you implemented before.

and returns the following:

- `first_ink_rows` : A numpy array with the shape `(N,)`, and the `int64` dtype.
  - `first_ink_rows[i]` is the index of the first row containing any ink pixel in the `i` th image. The indices should be **zero-based**.

```
In [26]:
def get_first_ink_row_index(is_row_inky):
    """
     Finds the first row containing ink pixels

        Parameters:
                is_row_inky (np.array): A numpy array with the shape (N, height), and
                This is the output of the get_is_row_inky function that you implemente

        Returns:
                first_ink_rows (np.array): A numpy array with the shape (N,), and the
    """
    first_ink_rows = is_row_inky.argmax(axis=1)
    #In case of multiple occurrences of the maximum values, the indices corresponding

    return first_ink_rows
```

```
In [27]:
# Performing sanity checks on your implementation
assert np.array_equal(get_first_ink_row_index(get_is_row_inky(train_images_threshed[:
                      np.array([5, 4, 5, 5, 7, 5, 4, 5, 5, 4]))
```

```
In [28]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [29]:
# Checking against the pre-computed test database
test_results = test_case_checker(get_first_ink_row_index, task_id=4)
assert test_results['passed'], test_results['message']
```

0.3.3.2 Getting the First Inky Columns

Since `is_row_inky` and `is_col_inky` have the same data structure, we could re-use the `get_first_ink_row_index` to define a corresponding function `get_first_ink_col_index` for columns; both of them have the same functionality and are essentially the same.

```
In [30]:
def get_first_ink_col_index(is_col_inky):
    return get_first_ink_row_index(is_col_inky)
```

0.3.4.1 Getting the Last Inky Rows

Task 5

Write the function `get_last_ink_row_index` that finds the last row containing ink pixels and takes following the arguments:

- `is_row_inky` : A numpy array with the shape `(N, height)`, and the `bool` dtype. This is the output of the `get_is_row_inky` function that you implemented before.

and returns the following:

- `last_ink_rows` : A numpy array with the shape `(N,)`, and the `int64` dtype.
  - `last_ink_rows[i]` is the index of the last row containing any ink pixel in the `i` th image. The indices should be **zero-based**.

Notes:

last column eg np.where(np.count_nonzero(x, axis=0)==0, -1, (x.shape[0]-1) - np.argmax(x[::-1,:]!=0, axis=0))

condition count number of non-zero values in each column and set = 1 if 0, otherwise 0 condition = np.count_nonzero(x, axis=0)==0

x if no non-zero values, -1, else, y x = -1

y if there are non-zero values rows - 1 - index of 1st occurence of max in each column of array when step backwards, all columns, where array values are 1 if O y = (x.shape[0]-1) - np.argmax(x[::-1,:]!=0, axis=0)

numpy.where(condition, [x, y,]/)

Return elements chosen from x or y depending on cond/ition. Parameters: condition: array_like, bool. Where True, yield x, otherwise yield y. x, y: array_like: Values from which to choose. x, y and condition need to be broadcastable to some shape.

Returns: out:ndarray - An array with elements from x where condition is True, and elements from y elsewhere.

```
In [31]:
def get_last_ink_row_index(is_row_inky):
    """
    Finds the last row containing ink pixels.

        Parameters:
                is_row_inky (np.array): A numpy array with the shape (N, height), and
                This is the output of the get_is_row_inky function that you implemente

        Returns:
                last_ink_rows (np.array): A numpy array with the shape (N,), and the i
    """

    last_ink_rows = (is_row_inky.shape[1]-1) - np.argmax(is_row_inky[:,::-1], axis=1)

    return last_ink_rows
```

```
In [32]:
# Performing sanity checks on your implementation
assert (get_last_ink_row_index(get_is_row_inky(train_images_threshed[:10,:,:])) ==
        np.array([24, 23, 24, 24, 26, 22, 23, 24, 24, 23])).all()

# Checking against the pre-computed test database
test_results = test_case_checker(get_last_ink_row_index, task_id=5)
assert test_results['passed'], test_results['message']
```

```
In [33]: # This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

0.3.4.2 Getting the Last Inky Columns

Since `is_row_inky` and `is_col_inky` have the same data structure, we could re-use the `get_last_ink_row_index` to define a corresponding function `get_last_ink_col_index` for columns; both of them have the same functionality and are essentially the same.

```
In [34]:
def get_last_ink_col_index(is_col_inky):
    return get_last_ink_row_index(is_col_inky)
```

0.3.5 The Final "Bounding Box" Pre-processor

Task 6

Write the function `get_images_bb` that applies the "Bounding Box" pre-processing step and takes the following arguments:

- `images` : A numpy array with the shape `(N,height,width)`, where
  - `N` is the number of samples and could be anything,
  - `height` is each individual image's height in pixels (i.e., number of rows in each image),
  - and `width` is each individual image's width in pixels (i.e., number of columns in each image).

    Do not assume anything about `images`'s dtype or number of samples.
- `bb_size` : A scalar with the default value of 20, and represents the desired bounding box size.

and returns the following:

- `images_bb` : A numpy array with the shape `(N,bb_size,bb_size)`, and the same dtype as `images`.

We have provided a template function that uses the previous functions and only requires you to fill in the missing parts. It also handles the input shapes in an agnostic way.

Important Note: Make sure that you use the `np.roll` function for this implementation.

Clarification with Example: Here we will describe in exact details how to produce the output image.

Let's focus on a single raw image, and let's call it `X`. We wish to produce the output image `Y`. Let's assume `X` has a `height` and a `width` of 28, and `bb_size` is 20. This means that `X` has a shape of `(28,28)` and `Y` has a shape of `(20,20)`. As a visual example, we'll assume that `X` is the left image (i.e., the raw image) shown below, and `Y` is the right image (i.e., the solution image).

alt text

Let's define the first/last inky rows with an example:

- The **first inky row** of `X` has an index of r_1. In the picture example, we have $r_1 = 6$. This means that `X[5,:]` has no ink in it, and `X[6,:]` has some ink elements in it.
- The **last inky row** of `X` has an index of r_2. In the picture example, we have $r_2 = 25$. This means that `X[25,:]` has some ink in it, and `X[26,:]` has no ink in it.

Let's define the first/last inky columns in a similar manner:

- The **first inky column** of `X` has an index of c_1. In the picture example, we have $c_1 = 5$. This means that `X[:,4]` has no ink in it, and `X[:,5]` has some ink elements in it.
- The **last inky column** of `X` has an index of c_2. In the picture example, we have $c_2 = 20$. This means that `X[:,20]` has some ink in it, and `X[:,21]` has no ink in it.

Now let's define the **inky middle row/column** of the raw image.

- The **inky middle row** of the raw image is $r_m = \lfloor \frac{r_1+r_2+1}{2} \rfloor$. In this example, we have $r_m = 16$, which is also shown in the picture.
- The **inky middle column** of the raw image is $c_m = \lfloor \frac{c_1+c_2+1}{2} \rfloor$. In this example, we have $c_m = 13$, which is also shown in the picture.

The middle row index of the output image is $r_{out} = \lfloor \frac{\text{bb\_size}}{2} \rfloor$. Similarly, we have the middle column index of the output image $c_{out} = \lfloor \frac{\text{bb\_size}}{2} \rfloor$. In this example, we have $r_{out} = c_{out} = 10$, which are marked with blue boxes in the solution image.

The **middle inky pixel of the raw image** is `X[r_m, c_m]`. This middle inky pixel is colored red in the raw image for clarification.

The **middle inky pixel of the solution image** is `Y[r_out, c_out]`. This middle inky pixel is colored red in the solution image for clarification

You must shift the raw image in a way that the **middle inky pixel of the raw image** gets placed on the **middle inky pixel of the solution image**. In other words, the red pixels should be placed on top of each other.

You will also have to cut some of rows/columns of the solution image properly to make sure it would have a shape of `(bb_size, bb_size)`. Furthermore, this whole discussion was for a single image, but you will have to take care of a batch of images as input, and produce a batch of bounded-box images as an output, and make sure all the dimensions/shapes work out properly.

```
Applies the "Bounding Box" pre-processing step to images.

    Parameters:
            images (np.array): A numpy array with the shape (N,height,width)
            bb_size: cols and rows
    Returns:
            images_bb (np.array): A numpy array with the shape
(N,bb_size,bb_size),
            and the same dtype as images.

    Apply the "Bounding Box" pre-processing step and takes the following
arguments:

    The template below uses the previous functions and handles the input
shapes in an agnostic way.
    Just fill in the missing parts.

    !!!Important: Use the np.roll function for this implementation.!!!

    numpy.roll(a, shift, axis=None)
    Roll array elements along a given axis.

    To roll is to shift elements such that elements shifted beyond the last
index on an axis in an array are re-introduced at the first index.

    Parameters
    a: array_like Input array.

    shift: int or tuple of ints
    The number of places by which elements are shifted.
    If tuple: axis must be a tuple of the same size, and each of the given
axes is shifted by the corresponding number.
            If int and axis is a tuple of ints: the same value is used for all
given axes.

    axis: int or tuple of ints, optional
    Axis or axes along which elements are shifted.
    Default:array is flattened before shifting, after which the original shape
is restored.

    Returns
    res: ndarray. Output array, with the same shape as a.
```

```
In [35]:
def get_images_bb(images, bb_size=20):


    if len(images.shape)==2:
        # In case a 2d image was given as input, we'll add a dummy dimension to be co
        images_ = images.reshape(1,*images.shape)
    else:
        # Otherwise, we'll just work with what's given
        images_ = images

    is_row_inky = get_is_row_inky(images_)
    is_col_inky = get_is_col_inky(images_)

    first_ink_rows = get_first_ink_row_index(is_row_inky) #INDEX of first inky row. S
    last_ink_rows = get_last_ink_row_index(is_row_inky)
    first_ink_cols = get_first_ink_col_index(is_col_inky)
    last_ink_cols = get_last_ink_col_index(is_col_inky)


    #Start my code here

    rows_mid = (first_ink_rows + last_ink_rows + 1) // 2
    cols_mid = (first_ink_cols + last_ink_cols + 1) // 2

    #shift = final - initial
    row_shift = bb_size // 2 - rows_mid
    col_shift = bb_size // 2 - cols_mid

    #images (np.array): A numpy array with the shape (N, height, width)
    #cut immages_.shape from (N, height, width) to (N, bb.size, bb.size)
    #assuming this is a reduction
    #want use slicing to return only begining to bb.size
    #np.reshape => np.resize => indexing should be used to reduce axes of arrays

#    images_bb = []

#    for i, img in enumerate(images):
#        #shift
#        img = np.roll(img, (row_shift[i], col_shift[i]), axis=(0, 1))
#        #cut
#        img = img[:bb_size, :bb_size]
#        images_bb.append(img)
#    images_bb = np.array(images_bb)

    ###
    #OR
    dtype = images_.dtype
    #create an empty array using
    images_bb = np.empty((images_.shape[0], bb_size, bb_size), dtype=dtype)
    #populate it one image at a time using something like:

    for i, img in enumerate(images):
        #shift
        img = np.roll(img, (row_shift[i], col_shift[i]), axis=(0, 1))
        #cut
        img = img[:bb_size, :bb_size]
```

```
        #insert
        images_bb[i,:,:] = img

    ###

    #End of my code here

    if len(images.shape)==2:
        # In case a 2d image was given as input, we'll get rid of the dummy dimension
        return images_bb[0]
    else:
        # Otherwise, we'll just work with what's given
        return images_bb
```

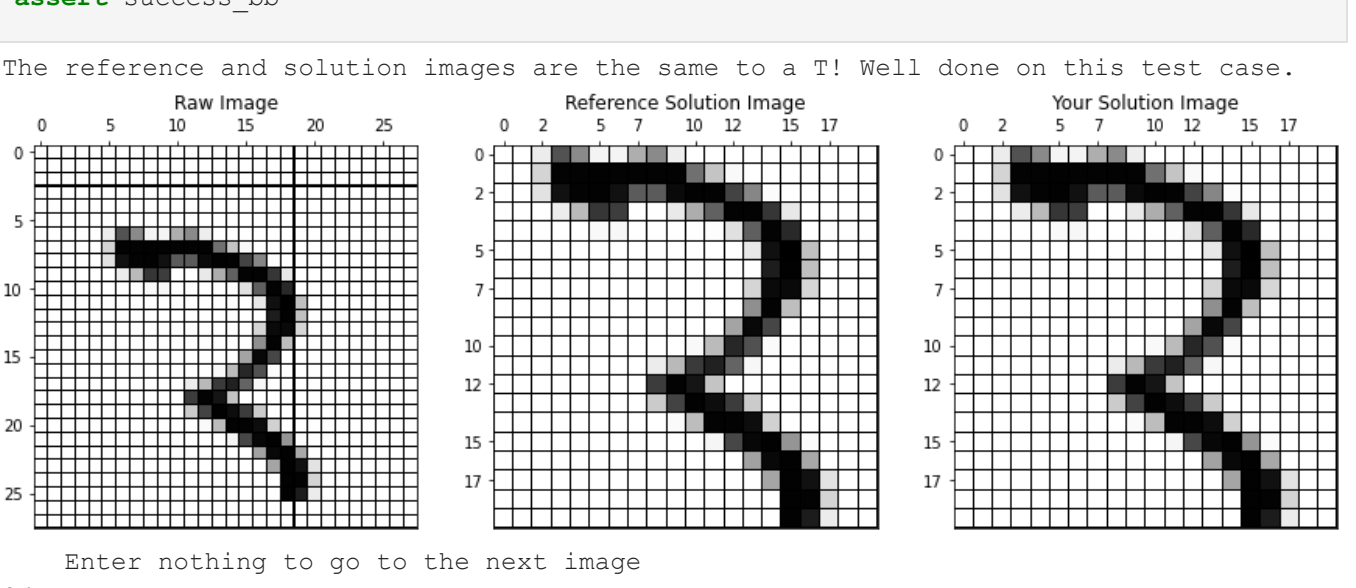
In [36]:

```
# This cell is left empty as a seperator. You can leave this cell as it is, and you sh
```

In [37]:

```
(orig_image, ref_image, test_im, success_bb) = show_test_cases(get_images_bb, task_id=

assert success_bb
```

```
The reference and solution images are the same to a T! Well done on this test case.
```




```
    Enter nothing to go to the next image
or
    Enter "s" when you are done to recieve the three images.
        **Don't forget to do this before continuing to the next step.**
s
```

Warning:

Do not leave the previous cell hanging; unless you enter "s" to stop it, you cannot evaluate other cells.

In [38]:

```
# This cell is left empty as a seperator. You can leave this cell as it is, and you sh
```

In [39]:

```
# Checking against the pre-computed test database
test_results = test_case_checker(get_images_bb, task_id=6)
assert test_results['passed'], test_results['message']
```

In [40]:

```
train_images_bb = get_images_bb(train_images_threshed)
eval_images_bb = get_images_bb(eval_images_threshed)
```

0.3.6 The Final "Stretched Bounding Box" Pre-processor

Task 7

Similarly, write the function `get_images_sbb` that applies the "Stretched Bounding Box" pre-processing step and takes following the arguments:

- `images` : A numpy array with the shape `(N,height,width)` , where
    - `N` is the number of samples and could be anything,
    - `height` is each individual image's height in pixels (i.e., number of rows in each image),
    - and `width` is each individual image's width in pixels (i.e., number of columns in each image).

    Do not assume anything about `images` 's dtype or number of samples.
- `bb_size` : A scalar with the default value of 20, and represents the desired bounding box size.

and returns the following:

- `images_sbb` : A numpy array with the shape `(N,bb_size,bb_size)` , and the same dtype and the range of values as `images` .

The `get_images_sbb` should find a tight-canvas of the inky area in each input image, and stretch it out to fill the full height and width of the output bounding-box. Please see the visual example in the **Assignment Summary** section; the right image should supposedly be the `get_images_sbb` function's output.

We have provided a template function that uses the previous functions and only requires you to fill in the missing parts. It also handles the input shapes in an agnostic way.

Hint: Make sure that you use the `skimage.transform.resize` function from the skimage library. Read about it at https://scikit-image.org/docs/dev/api/skimage.transform.html?highlight=resize#skimage.transform.resize. You may need to pay attention to the `preserve_range` argument.

skimage.transform.resize(image, output_shape, order=None, mode='reflect', cval=0, clip=True, preserve_range=False, anti_aliasing=None, anti_aliasing_sigma=None) **Resize image to match a certain size.**

Performs interpolation to up-size or down-size N-dimensional images. Note that anti-aliasing should be enabled when down-sizing images to avoid aliasing artifacts. For downsampling with an integer factor also see skimage.transform.downscale_local_mean.

Parameters image: ndarray. Input image.

output_shape iterable. Size of the generated output image (rows, cols[, ...][, dim]). If dim is not provided, the number of channels is preserved. In case the number of input channels does not equal the number of output channels a n-dimensional interpolation is applied.

Channels??

Returns resized. ndarray. Resized version of the input.

In [41]:

```
from skimage.util import img_as_uint

def get_images_sbb(images, bb_size=20):
    """
    Applies the "Stretched Bounding Box" pre-processing step to images.

        Parameters:
                images (np.array): A numpy array with the shape (N,height,width)

        Returns:
                images_sbb (np.array): A numpy array with the shape (N,bb_size,bb_size
                and the same dtype and the range of values as images.
    """

    if len(images.shape)==2:
        # In case a 2d image was given as input, we'll add a dummy dimension to be con
        images_ = images.reshape(1,*images.shape)
    else:
        # Otherwise, we'll just work with what's given
        images_ = images

    is_row_inky = get_is_row_inky(images)
    is_col_inky = get_is_col_inky(images)

    first_ink_rows = get_first_ink_row_index(is_row_inky)
    last_ink_rows = get_last_ink_row_index(is_row_inky)
    first_ink_cols = get_first_ink_col_index(is_col_inky)
    last_ink_cols = get_last_ink_col_index(is_col_inky)

    #Start my code here

    images_sbb = []

    for i, img in enumerate(images):
        #find a tight-canvas of the inky area in each input image,:
        img = img[first_ink_rows[i]:last_ink_rows[i] + 1, first_ink_cols[i]:last_ink_c
        #and stretch it out to fill the full height and width of the output bounding-b
        img = resize(img, output_shape=(bb_size,bb_size), preserve_range=True)
        images_sbb.append(img)
    images_sbb = np.array(images_sbb).astype(images_.dtype)

    #End of my code here


    if len(images.shape)==2:
        # In case a 2d image was given as input, we'll get rid of the dummy dimension
        return images_sbb[0]
    else:
        # Otherwise, we'll just work with what's given
        return images_sbb
```

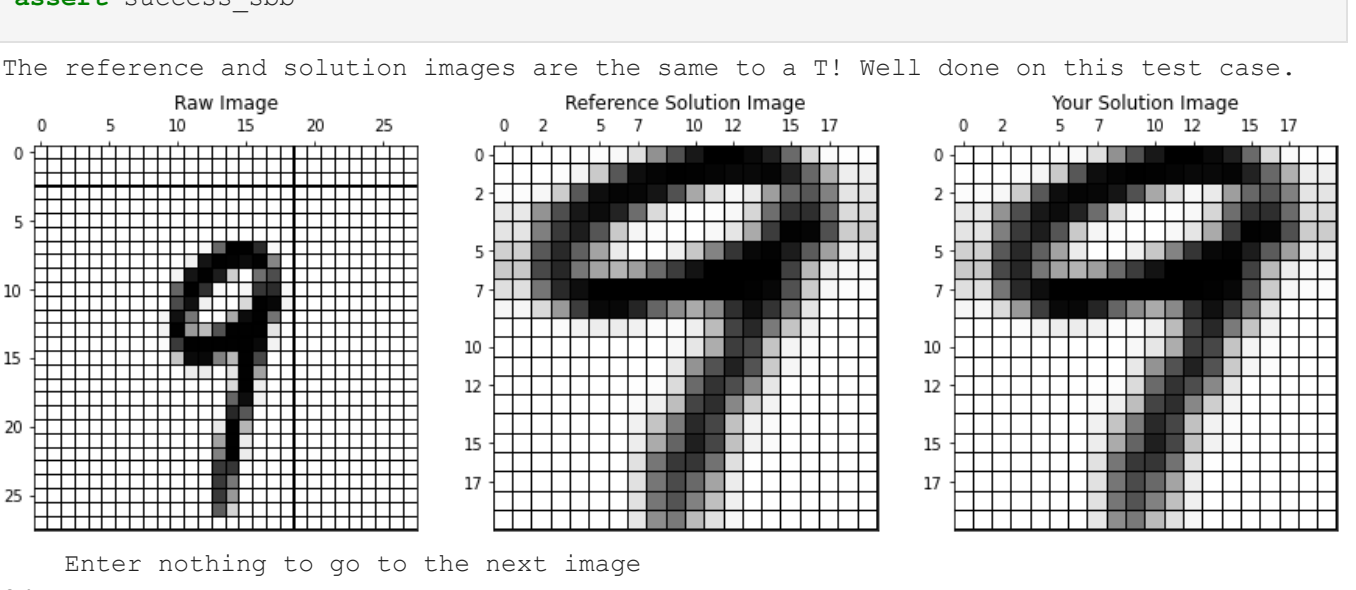
In [42]:

```
# This cell is left empty as a seperator. You can leave this cell as it is, and you sh
```

In [43]:

```
(orig_image, ref_image, test_im, success_sbb) = show_test_cases(get_images_sbb, task_

assert success_sbb
```

```
The reference and solution images are the same to a T! Well done on this test case.
```




```
    Enter nothing to go to the next image
or
    Enter "s" when you are done to recieve the three images.
        **Don't forget to do this before continuing to the next step.**
s
```

Warning:

Do not leave the previous cell hanging; unless you enter "s" to stop it, you cannot evaluate other cells.

In [44]:

```
# This cell is left empty as a seperator. You can leave this cell as it is, and you sh
```

In [45]:

```
# Checking against the pre-computed test database
test_results = test_case_checker(get_images_sbb, task_id=7)
assert test_results['passed'], test_results['message']
```

In [46]:

```
if perform_computation:
    print('This is gonna take a while to finish...')
    %time train_images_sbb = get_images_sbb(train_images_threshed)
    %time eval_images_sbb = get_images_sbb(eval_images_threshed)
```

```
This is gonna take a while to finish...
CPU times: user 39.1 s, sys: 14.1 s, total: 53.2 s
Wall time: 26.7 s
CPU times: user 6.3 s, sys: 2.29 s, total: 8.59 s
Wall time: 4.31 s
```

1. Naive Bayes Performances

Task 8

Similarly, write the function `train_nb_eval_acc` that trains Naive Bayes models and takes following the arguments:

- `train_images` : A numpy array with the shape `(N,height,width)` , where
    - `N` is the number of samples and could be anything,
    - `height` is each individual image's height in pixels (i.e., number of rows in each image),
    - and `width` is each individual image's width in pixels (i.e., number of columns in each image).

    Do not assume anything about `images` 's dtype or number of samples.
- `train_labels` : A numpy array with the shape `(N,)` , where `N` is the number of samples and has the `int64` dtype.
- `eval_images` : The evaluation images with similar characteristics to `train_images` .
- `eval_labels` : The evaluation labels with similar characteristics to `train_labels` .
- `density_model` : A string that is either `'Gaussian'` or `'Bernoulli'` . In the former (resp. latter) case, you should train a Naive Bayes with the Gaussian (resp. Bernoulli) density model.

and returns the following:

- `eval_acc` : a floating number scalar between 0 and 1 that represents the accuracy of the trained model on the evaluation data.

We have provided a template function that uses the previous functions and only requires you to fill in the missing parts. It also handles the input shapes in an agnostic way.

Note: You do not need to implement the Naive Bayes classifier from scratch in this assignment; Make sure you use `scikit-learn` 's Naive Bayes module for training and prediction in this task. We have already imported these two functions in the first code cell:

- `from sklearn.naive_bayes import GaussianNB, BernoulliNB`

In [47]:

```
def train_nb_eval_acc(train_images, train_labels, eval_images, eval_labels, density_mo
    """
    Trains Naive Bayes models, apply the model, and return an accuracy.

        Parameters:
                train_images (np.array): A numpy array with the shape (N,height,width)
                train_labels (np.array): A numpy array with the shape (N,), where N is
                has the int64 dtype.
                eval_images (np.array): The evaluation images with similar characteris
                eval_labels (np.array): The evaluation labels with similar characteris
                density_model (string): A string that is either 'Gaussian' or 'Bernoul

        Returns:
                eval_acc (np.float): a floating number scalar between 0 and 1 that
                represents the accuracy of the trained model on the evaluation data.
    """

    assert density_model in ('Gaussian', 'Bernoulli')

    # code start

    if density_model == 'Gaussian':
        clf = GaussianNB()
    else:
        clf = BernoulliNB()
    # why not:
#     clf = lambda density_model : GaussianNB() if density_model == 'Gaussian' else Be

    train_images = train_images.reshape(train_images.shape[0], -1)
    eval_images = eval_images.reshape(eval_images.shape[0], -1)

    # Train model X_train, Y_train
    clf.fit(train_images, train_labels)

    eval_acc = clf.score(X = eval_images, y = eval_labels)

    #code end

    return eval_acc

# Don't mind the following lines and do not change them
train_nb_eval_acc_gauss = lambda *args, **kwargs: train_nb_eval_acc(*args, density_mod
train_nb_eval_acc_bern = lambda *args, **kwargs: train_nb_eval_acc(*args, density_mode
```

In [48]:

```
# This cell is left empty as a seperator. You can leave this cell as it is, and you sh
```

In [49]:

```
# Checking against the pre-computed test database
test_results = test_case_checker(train_nb_eval_acc_gauss, task_id='8_G')
assert test_results['passed'], test_results['message'] # Gaussian Model Test Results

test_results = test_case_checker(train_nb_eval_acc_bern, task_id='8_B')
assert test_results['passed'], test_results['message'] # Bernoulli Model Test Results
```

In [50]:

```
df = None
if perform_computation:
    acc_nbg_thr = train_nb_eval_acc(train_images_threshed, train_labels,
                                    eval_images_threshed, eval_labels, density_model=
    acc_nbb_thr = train_nb_eval_acc(train_images_threshed, train_labels,
                                    eval_images_threshed, eval_labels, density_model=
    acc_nbg_sbb = train_nb_eval_acc(train_images_sbb, train_labels,
                                    eval_images_sbb, eval_labels, density_model='Gauss
    acc_nbb_sbb = train_nb_eval_acc(train_images_sbb, train_labels,
                                    eval_images_sbb, eval_labels, density_model='Berno

    df = pd.DataFrame([('Untouched images', acc_nbg_thr, acc_nbb_thr),
                       ('Stretched bounding box', acc_nbg_sbb, acc_nbb_sbb)
                      ], columns = ['Accuracy' , 'Gaussian', 'Bernoulli'])

df
```

Out[50]:

	Accuracy	Gaussian	Bernoulli
0	Untouched images	0.5491	0.8430
1	Stretched bounding box	0.8253	0.8098

2. Decision Forests Performances

Task 9

Similarly, write the function `train_tree_eval_acc` that trains Decision Forest models and takes following the arguments:

- `train_images` : A numpy array with the shape `(N,height,width)` , where
    - `N` is the number of samples and could be anything,
    - `height` is each individual image's height in pixels (i.e., number of rows in each image),
    - and `width` is each individual image's width in pixels (i.e., number of columns in each image).

    Do not assume anything about `images` 's dtype or number of samples.
- `train_labels` : A numpy array with the shape `(N,)` , where `N` is the number of samples and has the `int64` dtype.
- `eval_images` : The evaluation images with similar characteristics to `train_images` .
- `eval_labels` : The evaluation labels with similar characteristics to `eval_labels` .
- `tree_num` : An integer number representing the number of trees in the dicision forest.
- `tree_depth` : An integer number representing the maximum tree depth in the dicision forest.
- `random_state` : An integer with a default value of 12345 that should be passed to the scikit-learn's classifer constructor for reproducibility and auto-grading (**Do not assume** that it is always 12345).

and returns the following:

- `eval_acc` : A floating number scalar between 0 and 1 that represents the accuracy of the trained model on the evaluation data.

We have provided a template function that uses the previous functions and only requires you to fill in the missing parts. It also handles the input shapes in an agnostic way.

Note: You do not need to implement the Random Forest classifier from scratch in this assignment; Make sure you use `scikit-learn` 's Random Forest module for training and prediction in this task. We have already imported this function in the first code cell:

- `from sklearn.ensemble import RandomForestClassifier`
- You may need to set "shuffle = True" due to a known sklearn issue.

clf = RandomForestClassifier(max_depth=2, random_state=0) clf.fit(X, y)

Parameters (n_estimators=100 max_depth=None random_state=None

Methods

- apply(X): Apply trees in the forest to X, return leaf indices.
- fit(X, y): Build a forest of trees from the training set (X, y). Parameters: X "(n_samples, n_features)" y "array-like of shape (n_samples,)..."
- score(X, y)Return the mean accuracy on the given test data and labels.
- `random_state` : An integer with a default value of 12345 that should be passed to the scikit-learn's classifer constructor for reproducibility and auto-grading (**Do not assume** that it is always 12345).
- You may need to set "shuffle = True" due to a known sklearn issue.

In [51]:

```
def train_tree_eval_acc(train_images, train_labels, eval_images, eval_labels, tree_num
    """
    Trains Naive Bayes models, apply the model, and return an accuracy.

        Parameters:
                train_images (np.array): A numpy array with the shape (N,height,width)
                train_labels (np.array): A numpy array with the shape (N,), where N is
                has the int64 dtype.
                eval_images (np.array): The evaluation images with similar characteris
                eval_labels (np.array): The evaluation labels with similar characteris
                tree_num (int): An integer number representing the number of trees in
                tree_depth (int): An integer number representing the maximum tree dept
                random_state (int): An integer with a default value of 12345 that shou
                the scikit-learn's classifer constructor for reproducibility and auto-

        Returns:
                eval_acc (np.float): a floating number scalar between 0 and 1 that
                represents the accuracy of the trained model on the evaluation data.
    """

    tree_num = int(tree_num)
    tree_depth = int(tree_depth)
    random_state = int(random_state)

    # student code beginning
    train_images = train_images.reshape(train_images.shape[0], -1)
    eval_images = eval_images.reshape(eval_images.shape[0], -1)

    clf = RandomForestClassifier(max_depth=tree_depth, n_estimators=tree_num, random_s
    clf.fit(X=train_images, y=train_labels)
    eval_acc = clf.score(X = eval_images, y=eval_labels)
    #student code end

    return eval_acc
```

In [52]:

```
# This cell is left empty as a seperator. You can leave this cell as it is, and you sh
```

In [53]:

```
# Checking against the pre-computed test database
test_results = test_case_checker(train_tree_eval_acc, task_id=9)
assert test_results['passed'], test_results['message']
```

2.1 Accuracy on the Untouched Images

In [54]:

```
df = None
if perform_computation:
    tree_nums = [10, 20, 30]
    tree_depths = [4, 8, 16]

    train_images = train_images_threshed
    eval_images = eval_images_threshed
    acc_arr_unt = np.zeros((len(tree_nums), len(tree_depths)))
    for row, tree_num in enumerate(tree_nums):
        for col, tree_depth in enumerate(tree_depths):
            acc_arr_unt[row, col] = train_tree_eval_acc(train_images, train_labels, ev
                                                        tree_num=tree_num, tree_depth=

    df = pd.DataFrame([(f'#trees = {tree_num}', *tuple(acc_arr_unt[row])) for row, tre
                      columns = ['Accuracy'] + [f'depth={tree_depth}'for col, tree_dep

    print('Untouched Images:')
df
```

```
Untouched Images:
```

Out[54]:

	Accuracy	depth=4	depth=8	depth=16
0	#trees = 10	0.7496	0.8923	0.9489
1	#trees = 20	0.7707	0.9127	0.9585
2	#trees = 30	0.7883	0.9169	0.9630

2.2 Accuracy on the "Stretched Bounding Box" Images

In [55]:

```
df = None
if perform_computation:
    tree_nums = [10, 20, 30]
    tree_depths = [4, 8, 16]

    train_images = train_images_sbb
    eval_images = eval_images_sbb
    acc_arr_sbb = np.zeros((len(tree_nums), len(tree_depths)))
    for row, tree_num in enumerate(tree_nums):
        for col, tree_depth in enumerate(tree_depths):
            acc_arr_sbb[row, col] = train_tree_eval_acc(train_images, train_labels, ev
                                                        tree_num=tree_num, tree_depth=

    df = pd.DataFrame([(f'#trees = {tree_num}', *tuple(acc_arr_sbb[row])) for row, tre
                      columns = ['Accuracy'] + [f'depth = {tree_depth}'for col, tree_

    print('Stretched Bounding Box Images:')
df
```

```
Stretched Bounding Box Images:
```

Out[55]:

	Accuracy	depth = 4	depth = 8	depth = 16
0	#trees = 10	0.7419	0.9043	0.9527
1	#trees = 20	0.7715	0.9162	0.9639
2	#trees = 30	0.7879	0.9248	0.9671

2.3 Accuracy on the "Bounding Box" Images

In [56]:

```
df = None
if perform_computation:
    tree_nums = [10, 20, 30]
    tree_depths = [4, 8, 16]

    train_images = train_images_bb
    eval_images = eval_images_bb
    acc_arr_bb = np.zeros((len(tree_nums), len(tree_depths)))
    for row, tree_num in enumerate(tree_nums):
        for col, tree_depth in enumerate(tree_depths):
            acc_arr_bb[row, col] = train_tree_eval_acc(train_images, train_labels, eva
                                                       tree_num=tree_num, tree_depth=t

    df = pd.DataFrame([(f'#trees = {tree_num}', *tuple(acc_arr_bb[row])) for row, tree
                      columns = ['Accuracy'] + [f'depth = {tree_depth}'for col, tree_d

    print('Bounding Box Images:')
df
```

```
Bounding Box Images:
```

Out[56]:

	Accuracy	depth = 4	depth = 8	depth = 16
0	#trees = 10	0.7406	0.8865	0.9476
1	#trees = 20	0.7716	0.9050	0.9576
2	#trees = 30	0.7801	0.9089	0.9608

In [ ]: