Assignment:

- Perform handwriting recognition on the numerals 0-9, using two both naive Bayes and decision forest methods. Can you achieve a 95% accuracy?
- Use the MNIST dataset for training and test examples.
- Use skimage, sklearn, pandas, matplotlib, numpy, torchvision as necessary.

<u>Step 1: Import key libraries:</u>

```
%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 utils import show_test_cases, test_case_checker, perform_computation
```

<u>Step 2: Load Data:</u>

Since the MNIST data is stored in a binary format, use torchvision to load it easily. If we loaded it earlier, use previous copy instead of downloading it again. Load it into training and test (eval) sets.

```
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, 
target_transform=None, download=download_)
    data_eval = torchvision.datasets.MNIST('mnist', train=False, transform=None, 
target_transform=None, download=download_)

    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, 
train_labels=train_labels,
             eval_images_raw=eval_images_raw, eval_labels=eval_labels)
```

Check that we have the right number of training images and they are the right size.

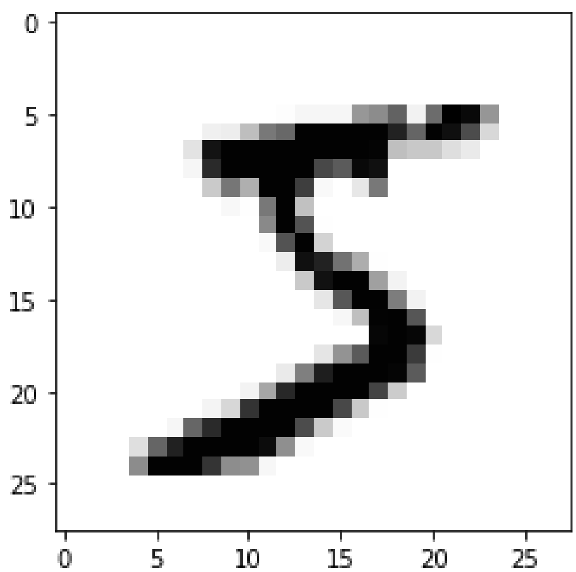
```
train_images_raw.shape
(60000, 28, 28)
```

That's correct, 60,000 images at a size of 28x28 pixels.

Lets look at one image, both numerically, and graphically.
Numerically:

Graphically:

```
<matplotlib.image.AxesImage at 0x7f6634d74700>
```



Thresholding:

We now want to convert the image to black and white instead of shades of grey. We'll use a threshold value for the pixels, and anything above that value will be set to black, and anything below it will be white.

```
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, 82, 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]
```

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

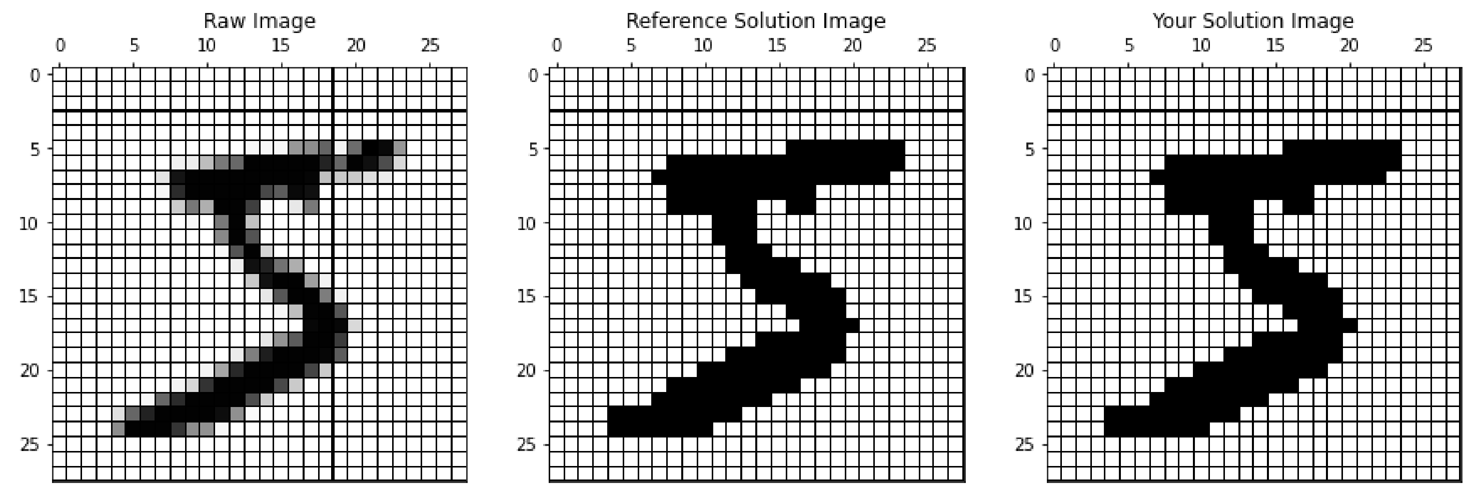
- `images_raw`: A numpy array. Do not assume anything about its shape, dtype or range of values. Your function should be careless about these attributes.
- `threshold`: A scalar value.

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 the value of `threshold`.

Test thresholding:

```
(orig_image, ref_image, test_im, success_thr) = show_test_cases(lambda x:
get_thresholded(x, 20), task_id='1_V')

assert success_thr
```




```
The reference and solution images match, the solution is correct.
```

```
def get_thresholded(images_raw, threshold):
    """
    Perform image thresholding.

        Parameters:
                images_raw (np,array): Do not assume anything about its
shape, dtype or range of values.
                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_raw, and the bool dtype.
                This array should indicate whether each elemelent of
images_raw is greater than or equal to
                threshold.

    threshed_image = images_raw >= threshold
    return threshed_image
```

Creating "Bounding Box" Images

We now need to create bounding boxes around the images to scale them to a uniform size. The first step here is to determine which columns and rows have color or "ink" in them.

Step 1: Find "Inky" rows

To do this: 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`)

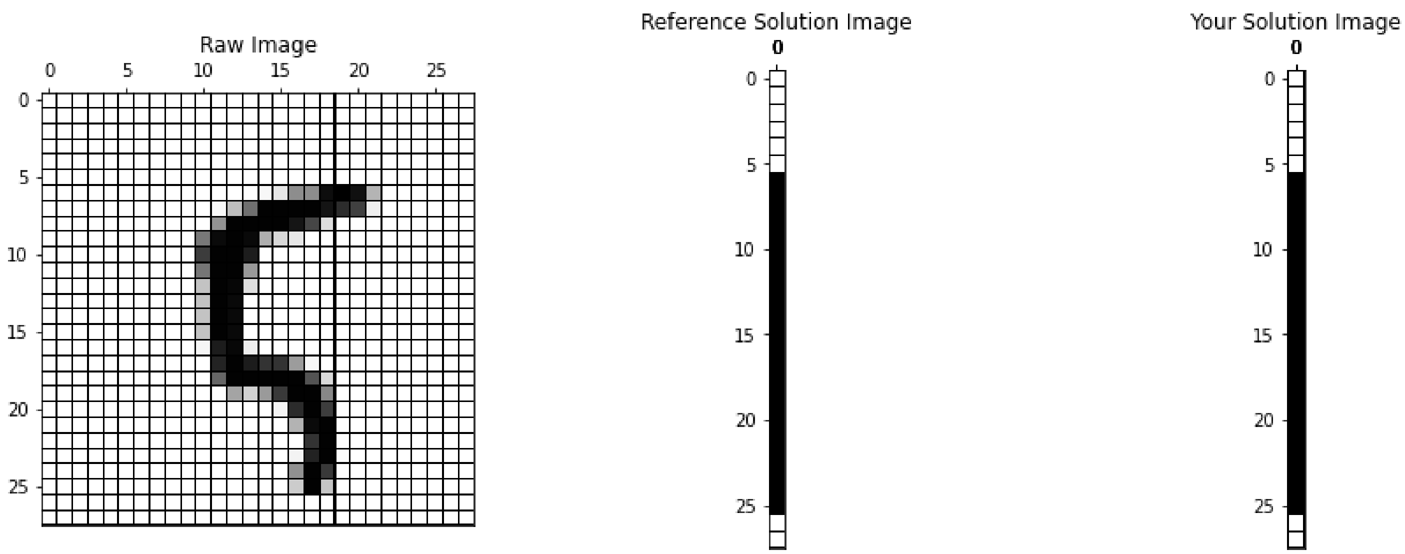
and returns the following:

- `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.

```
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 the bool dtype.
    """
    is_row_inky = np.sum(images, axis = 2) > 0

    return is_row_inky
```

Now we need to test if the rows are calculated correctly:

```
(orig_image, ref_image, test_im, success_is_row_inky) = show_test_cases(lambda x:
np.expand_dims(get_is_row_inky(x), axis=2),

task_id='2_V')

assert success_is_row_inky
```




```
The reference and solution images match, the solution is correct.
```

Step 2: Find "Inky" columns.

Similar to the inky rows, rind the inky columns

Similar to get_is_row_inky above, 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.)

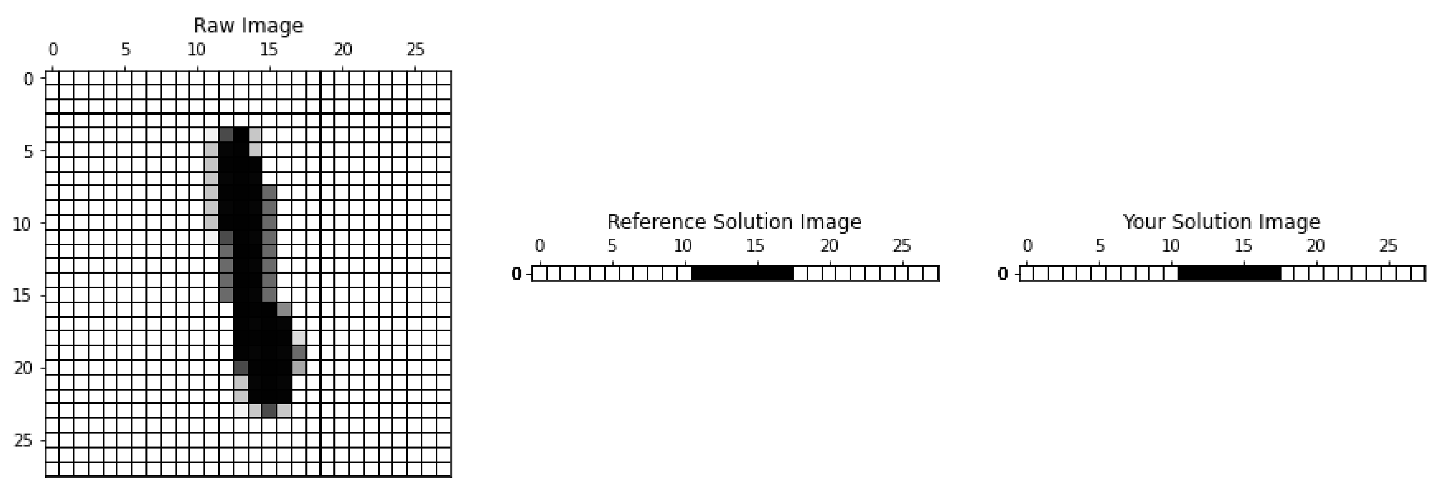
and returns the following:

- 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 jth column of the ith image was an ink pixel, and False otherwise.

```
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 the bool dtype.
    """

    # your code here

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

    return is_col_inky
```

Now lets test if the columns are calculated correctly:

```
(orig_image, ref_image, test_im, success_is_col_inky) = show_test_cases(lambda x:
np.expand_dims(get_is_col_inky(x), axis=1),

task_id='3_V')

assert success_is_col_inky
```




```
The reference and solution images match, the solution is correct.
```

Step 3: Now we need to find the first and last inky rows and columns, so we can determine where the image pixels are located in the grid.

(note that the for finding the first inky columns, the first inky rows routine can be re-used. The calculations are the same.)

```
def get_first_ink_col_index(is_col_inky):

    return get_first_ink_row_index(is_col_inky)
```

```
def get_first_ink_row_index(is_row_inky):

            Parameters:

                is_row_inky (np.array): 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.

        Returns:

                first_ink_rows (np.array): A numpy array with the shape (N,), and
the int64 dtype.

    """

    first_ink_rows = np.argmax(is_row_inky, axis = 1)

    return first_ink_rows
```

Finding the last inky columns and rows follow similarly:

```
def get_last_ink_row_index(is_row_inky):

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

    return last_ink_rows
```

```
def get_last_ink_col_index(is_col_inky):

    return get_last_ink_row_index(is_col_inky)
```

Bounding Boxes:

Now that we have found where the first and last inked columns and rows are, we can use bounding boxes to resize the image to fit a standard grid size.

Task:

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.

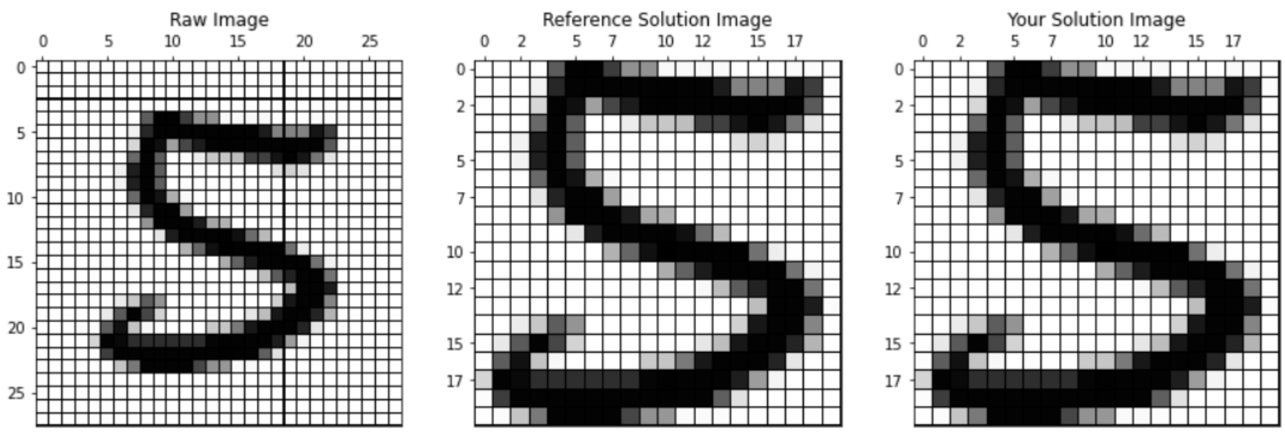
and returns the following:

- $images\_bb$: A numpy array with the shape (N, bb_size, bb_size), and the same dtype as $images$.

```
def get_images_bb(images, bb_size=20):
    """
         Parameters:
                images (np.array): A numpy array with the shape (N,height,width)
        Returns:
                images_bb (np.array): A numpy array with the shape
(N,bb_size,bb_size),
                and the same dtype as images.
    """
    if len(images.shape)==2:
        # In case a 2d image was given as input, we'll add a dummy dimension to
be consistent
        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)
    inky_middle_row = np.floor((first_ink_rows + last_ink_rows + 1) /
2).astype(int)   # calculate middle
    inky_middle_column = np.floor((first_ink_cols + last_ink_cols + 1) /
2).astype(int)    # calculate middle

    # calculate how much to shift rows and columns
    row_shift = np.int((bb_size)/ 2) - inky_middle_row
    col_shift = np.int((bb_size)/ 2) - inky_middle_column


    # create dummy arrays to be replaced with new image once we calculate it
    images_bb=np.zeros((images_.shape[0],bb_size,bb_size), dtype=np.uint8)

    # loop through images to shift each one then copy to images_bb array
    for i in range(images_.shape[0]):
        images_shifted_cols = np.roll(images_[i], col_shift[i], axis = 1)  #
first shift columns

        images_shifted_rows = np.roll(images_shifted_cols, row_shift[i], axis =
0)  # now shift rows

        images_bb[i]= images_shifted_rows[0:bb_size,0:bb_size]  # put shifted
image into images_bb array



    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
```

Now we'll test how the images are stretched in the bounding box:

```
(orig_image, ref_image, test_im, success_bb) =
show_test_cases(get_images_bb, task_id='6_V')


assert success_bb
```




```
The reference and solution images match, the solution is correct.
```

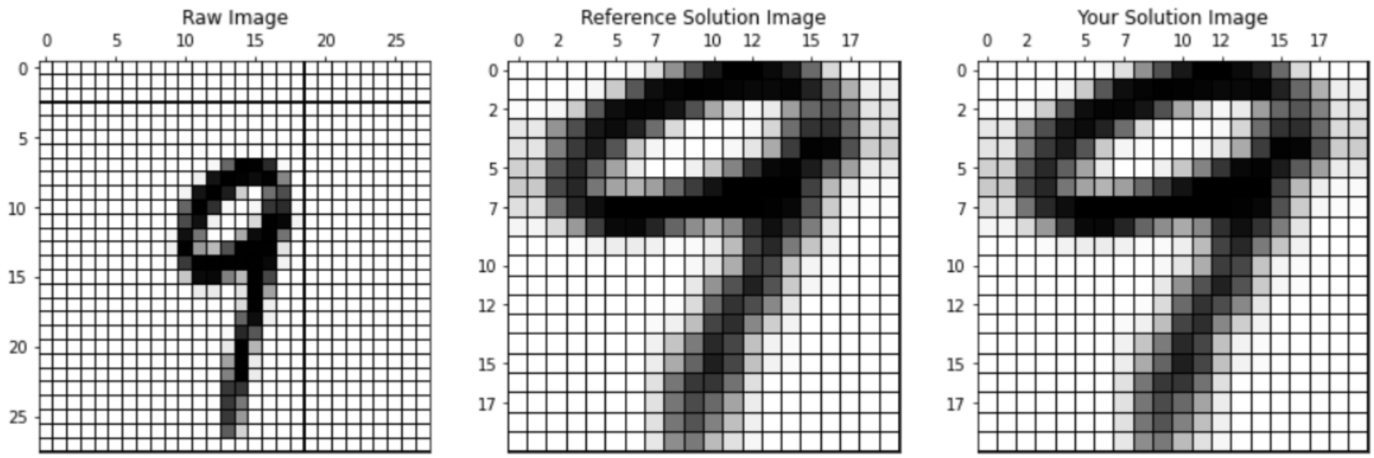
Stretched Bounding Boxes:

Another way to do the bounding box is to stretch them to take all the available boxes in the grid. This can help the image recognition.

```
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
consistent
        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)

    images_sbb = np.zeros((images_.shape[0],bb_size,bb_size), dtype=np.uint8)


    for i in range(images_.shape[0]):

        image_cropped = images_[i][first_ink_rows[i]:(last_ink_rows[i]+1),
first_ink_cols[i]:(last_ink_cols[i]+1)]

        images_sbb[i] = resize(image_cropped,(bb_size,bb_size), preserve_range = True)


    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
```

Testing the stretched bounding boxes:

```
(orig_image, ref_image, test_im, success_sbb) = show_test_cases(get_images_sbb,
task_id='7_V')

assert success_sbb
```




```
The reference and solution images match, the solution is correct.
```

Evaluating Performance:

To evaluate the accuracy, we'll train and test Naive Bayes models:

We'll write the train_nb_eval_acc function to do this, it will take the following 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 Naïve 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.

Instead of writing the Naive Bays function from scratch, we'll use the Naïve Bayes module in the scikit-learn library. We'll evaluate using both the GaussianNB and BernoulliNB versions.

```
def train_nb_eval_acc(train_images, train_labels, eval_images, eval_labels,
density_model='Gaussian'):
    """
    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 the number of samples and
                has the int64 dtype.
                eval_images (np.array): The evaluation images with similar
characteristics to train_images.
                eval_labels (np.array): The evaluation labels with similar
characteristics to train_labels.
                density_model (string): A string that is either 'Gaussian' or
'Bernoulli'.

        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')


    train_images_reshaped =
train_images.reshape((train_images.shape[0],train_images.shape[1]*train_images.sha
pe[2]))
    eval_images_reshaped =
eval_images.reshape((eval_images.shape[0],eval_images.shape[1]*eval_images.shape[2
]))


    if density_model == "Gaussian":
        trainz = GaussianNB()

        eval_acc =
trainz.fit(train_images_reshaped,train_labels).score(eval_images_reshaped,eval_lab
els)

    else:
        trainz = BernoulliNB()

        output =
trainz.fit(train_images_reshaped,train_labels).predict(eval_images_reshaped)
        eval_acc = np.mean(output == eval_labels
return eval_acc


train_nb_eval_acc_gauss = lambda *args, **kwargs: train_nb_eval_acc(*args,
density_model='Gaussian', **kwargs)

train_nb_eval_acc_bern = lambda *args, **kwargs: train_nb_eval_acc(*args,
density_model='Bernoulli', **kwargs)
```

Now testing the results:

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

    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
```

Accuracy results:

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

So the stretched bounding box works far better than the raw images for the Gaussian approach, but a little worse for Bernoulli.

Decision Forests:

Now lets test decision forests at various numbers of trees and tree depths.

We'll use the random forest classifier from scikit-learn:

```
def train_tree_eval_acc(train_images, train_labels, eval_images, eval_labels, tree_num=10,
tree_depth=4, random_state=12345):
    """
    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 the
number of samples and
                has the int64 dtype.
                eval_images (np.array): The evaluation images with similar characteristics
to train_images.
                eval_labels (np.array): The evaluation labels with similar characteristics
to train_labels.
                tree_num (int): An integer number representing the number of trees in the
decision forest.
                tree_depth (int): An integer number representing the maximum tree depth in
the decision forest.
                random_state (int): An integer with a default value of 12345 that should
be passed to
                the scikit-learn's classifer constructor for reproducibility and auto-
grading

        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)

    train_images_reshaped =
train_images.reshape((train_images.shape[0],train_images.shape[1]*train_images.shape[2]))
    eval_images_reshaped =
eval_images.reshape((eval_images.shape[0],eval_images.shape[1]*eval_images.shape[2]))

    treez = RandomForestClassifier(max_depth = tree_depth, random_state =
random_state, n_estimators = tree_num)

    results = treez.fit(train_images_reshaped,
train_labels).predict(eval_images_reshaped)

    eval_acc = treez.fit(train_images_reshaped,
train_labels).score(eval_images_reshaped, eval_labels)

return eval_acc
```

Accuracy on the Raw Images:

```
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,
eval_images, eval_labels,

                                                        tree_num=tree_num,
tree_depth=tree_depth, random_state=12345)



    df = pd.DataFrame([(f'#trees = {tree_num}', *tuple(acc_arr_unt[row])) for row,
tree_num in enumerate(tree_nums)],

                      columns = ['Accuracy'] + [f'depth={tree_depth}'for col,
tree_depth in enumerate(tree_depths)])

df
```

Raw Image Accuracy:

	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

Accuracy on the Bounding Box Images:

```
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,
eval_images, eval_labels,
                                                       tree_num=tree_num,
tree_depth=tree_depth, random_state=12345)

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

df
```

Bounding Box Image Accuracy:

	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

Accuracy on the Stretched Bounding Box Images:

```
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,
eval_images, eval_labels,
                                                        tree_num=tree_num,
tree_depth=tree_depth, random_state=12345)

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

df
```

Stretched Bounding Box Image Accuracy:

	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

In conclusion, the bounding box images did not result in better accuracy than the raw images, but the stretched bounding box images were a little better than the raw images. It turns out that greater than 95% accuracy can be achieved with a large enough tree depth. The tree depth appears to have more effect on the accuracy than the number of trees.