PROJECT MILESTONE

Team: reCAPTCHA
Members: Johann Ryan, Rithvik Subramanya, Indresh Srivastava, Dominic Grazioli

Work Planned for Week 8

Week 8:
- Separation of individual characters (completed) - 3 hours
- Features extracted and neural network setup - 3 hours

Work Completed in Week 8

Week 8:
- Fix to character separation algorithm
- Setup CNN to run on easy captcha data
- SVM dataset setup

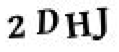
Project Plan for Upcoming Weeks

Week 9:
- Run SVM on dataset created
- Train neural network on easy captcha data
- Test our neural net on individual characters

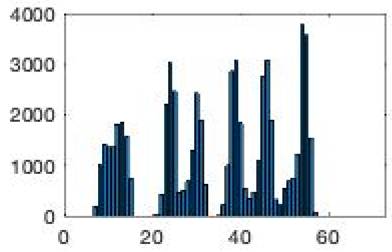
Week 10:
- Stitch individual character identification to identify all text in the captcha image
- Prepare for the presentation

Character Separation Fixes (Johann)

Prior to last week, we encountered a few bugs with our boundary box detection on the Captchas for the easy data set. In particular, there were some images where two of the characters were slightly overlapping. This caused the bwlabel to identify the conjoined characters under one region, which resulted in us not obtaining the required amount of characters (4) for that iteration. Hence, to solve this issue we needed to take a more customized approach. Thus, we worked on deriving an algorithm that could somewhat tell where the spaces in between characters were and split the image there. This was done by first summing up the columns of the image where each column value was subtracted from 255. As a result, columns on the image where there were no black pixels (i.e. no parts of characters) had values of zero whereas columns that included parts of characters had much higher values. To visualize this, consider the following image from our dataset:



This image is an example of a case where simply using bwlabel doesn't work since the H and J characters are close enough that the edge detection would conjoin them. However, by obtaining our modified column sums, we can acquire the following data represented by the histogram below.



In this histogram, the x-axis represents the index of the column for the "2DHJ" image above while the y-axis represents how many black pixels there are in the column. As one can see from the histogram, there are 5 local minima in the data where the amount of black pixel's drop in that column. These refer to the gap between the 2 and D, the center of the D, the gap between the D and H, the center of the H, and the gap between the H and J. However, we only want the gaps, which almost always have close to 0 values on the histogram. Hence, we use the "islocalmin" function, which outputs 0 if the column is not a local minimum and 1 if it is. We then subtract this by 1 and multiply by -9999 so that all the 0's turn into 9999 and all the 1's turn into 0's. We then take this and add it to histogram values so that any value that isn't a local minimum becomes extremely large. With this new weighting, we can then take the indexes at the 3 smallest values and use that as our splitting points. Also, note that this includes the gap between H and J since although that value isn't exactly zero, it is very small due there only being a black/gray pixel at that column. Furthermore, this algorithm doesn't select the many zeros in the histogram over that point since only 2 of the many zeros are chosen as local minimums.

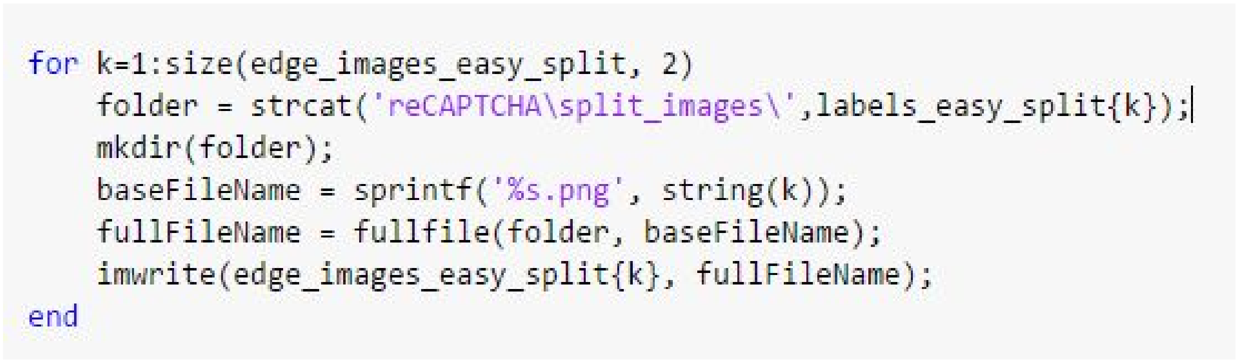
Once we have the split characters from the images, we can take bounding boxes around each of the characters and reshape them to uses as features. However, one thing we observed when using the bounding box functionality from regionprops is that it tending to over "crop" the images in some cases. Hence, we relied on our own implementation of the bounding box where we shaved away all the column sums and row sums that were zero.

While this approach is fairly reliable, it does still have a few issues/kinks to work out. For instance, there are some cases where two local minima are created when there is actually just one since a noise pixel split the minima into two. However, we expect that we can resolve these issues by incorporating distance thresholds into our algorithm. Moreover, this method could have exciting results on our hard dataset since we could incorporate thresholds on column sums to potentially divide characters even if there is noise or horizontal lines cutting through them. However, even in the event where this is unreliable, we can always use HOG filters to obtain meaningful features for use in a support vector machine or shallow neural network.

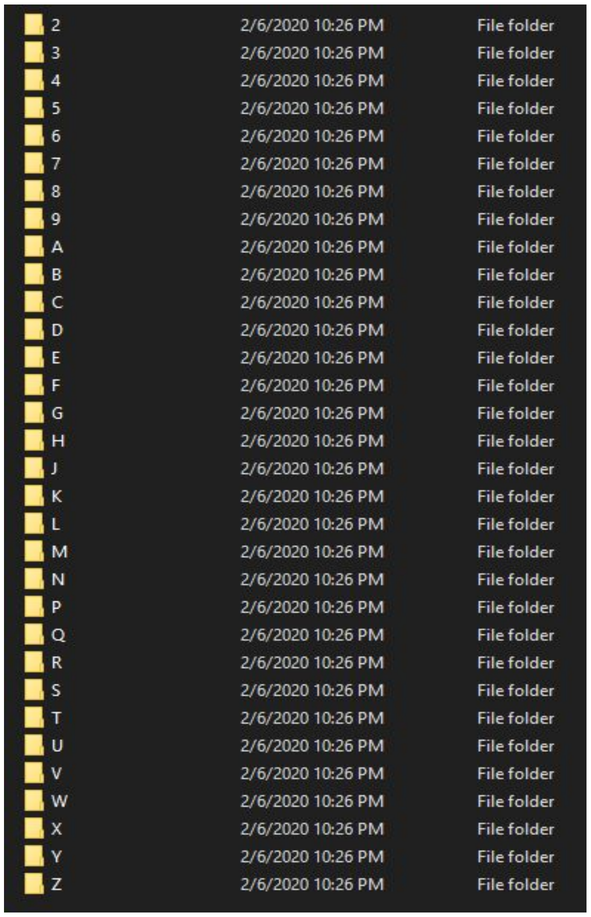
Another exciting approach we have considered using to split characters is to use k-means clustering. With k-means clustering where k is equal to the number of characters, we can potentially identify the centroids of the "densest" concentration black pixels (i.e. the characters) in the image. Once we have these centroids, we can then take the horizontal midpoint between them to decide where to split the images. Moreover, what makes this approach interesting is that it should be able to work somewhat well on even the hard dataset, as the characters in the images could be described as the densest areas of black pixels. Hence, in theory the k-means clustering should be able to tell where characters are approximately located, though obtaining bounding boxes of these characters might be more challenging (hopefully the convolutional neural networks can handle these without them). However, one thing we need to be careful of when using k-means clustering is that setting the initial centroids may play a crucial role in identifying characters. In other words, if we randomly/haphazardly set the initial centroids on the image, we could lock onto the wrong clusters. However, this can easily be mitigated by placing our initial centroids evenly along the horizontal axis. Furthermore, we may even want to consider using a modified k-means clustering where we only cluster based on the x-coordinate since we just want to know where to split the image and this might prevent us from having clusters vertically apart but close horizontally.

Easy Data CNN Setup

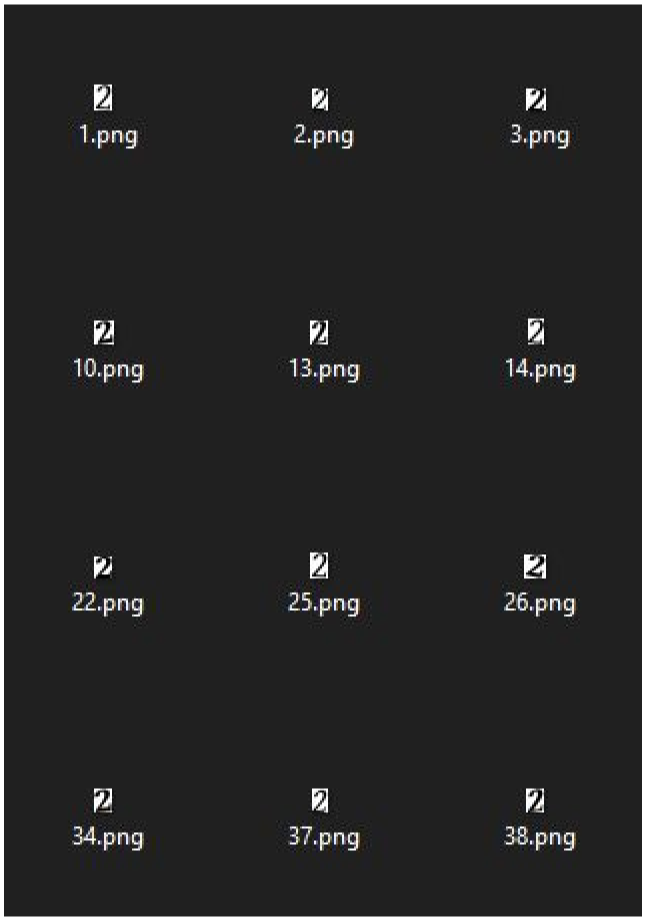
So last we left our images broken up into individual characters. We had two arrays, one holding the actual character image and the other holding the label for the character represented within that image. After we successfully separated the data we wanted to use one of the existing neural nets like the resnet or alexnet to train the net on the features extracted from these images. However, before we could input opur data into the neural net we needed to convert it into an image datastore (since that is the only form in which the net accepts the image data). For this we needed to export the image as a png and save these roughly 40000 character images along with their labels into a directory which could be passed to the imagedatastore function. To do this we found it challenging to figure out a way to unique save these images and use the labels within the filename to pull in for imagedatastore label use. We found a simple workaround to this problem, instead of labelling each image with its actual character label we instead grouped images with similar character representations under a common folder , labeled by the character value. The following code is what helped us achieve that.



```
for k=1:size(edge_images_easy_split, 2)
    folder = strcat('reCAPTCHA\split_images\',labels_easy_split{k});
    mkdir(folder);
    baseFileName = sprintf('%s.png', string(k));
    fullFileName = fullfile(folder, baseFileName);
    imwrite(edge_images_easy_split{k}, fullFileName);
end
```



2	2/6/2020 10:26 PM	File folder
3	2/6/2020 10:26 PM	File folder
4	2/6/2020 10:26 PM	File folder
5	2/6/2020 10:26 PM	File folder
6	2/6/2020 10:26 PM	File folder
7	2/6/2020 10:26 PM	File folder
8	2/6/2020 10:26 PM	File folder
9	2/6/2020 10:26 PM	File folder
A	2/6/2020 10:26 PM	File folder
B	2/6/2020 10:26 PM	File folder
C	2/6/2020 10:26 PM	File folder
D	2/6/2020 10:26 PM	File folder
E	2/6/2020 10:26 PM	File folder
F	2/6/2020 10:26 PM	File folder
G	2/6/2020 10:26 PM	File folder
H	2/6/2020 10:26 PM	File folder
J	2/6/2020 10:26 PM	File folder
K	2/6/2020 10:26 PM	File folder
L	2/6/2020 10:26 PM	File folder
M	2/6/2020 10:26 PM	File folder
N	2/6/2020 10:26 PM	File folder
P	2/6/2020 10:26 PM	File folder
Q	2/6/2020 10:26 PM	File folder
R	2/6/2020 10:26 PM	File folder
S	2/6/2020 10:26 PM	File folder
T	2/6/2020 10:26 PM	File folder
U	2/6/2020 10:26 PM	File folder
V	2/6/2020 10:26 PM	File folder
W	2/6/2020 10:26 PM	File folder
X	2/6/2020 10:26 PM	File folder
Y	2/6/2020 10:26 PM	File folder
Z	2/6/2020 10:26 PM	File folder



After this we plan to use these simple character images to create the imagedatastore and use it to train the neural net. Hopefully tomorrow in class we have our network trained with these basic character images.

SVM Dataset Setup

Using the images broken up into the different images, we separated the large dataset into different subsets that will be used for SVMs. We identified that we will be needing 36 SVMs, one for each letter of the alphabet in lowercase, and 10 for the digits. With this information, we needed to make 36 subsets of our dataset with a reasonable division between the character we are looking for and the characters we are not.