**Canny Digit Recognizer**

**ECE420 Final Project Report**

Joseph Kim (jdk5), Yufei Wang (yufeiw3)

05/02/2021

**Introduction**

In our assigned project lab, we have explored the canny edge detection algorithm. For our final project, we expand our work by developing an Android app that can handle canny edge detection and handwritten digit recognition. The app would take in an image input, process the image with the Canny Edge Detector, and be able to identify which of the 10 digits (0-9) is being represented.

**Background**

The edge detection paper describes several different algorithms for edge detection, but for the assigned lab and final project, we decided on the Canny edge detector. However, to build upon the Canny edge detector that we previously implemented in Python, we wanted to use that edge detector to process images for digit classification. In fact, edge detection is a fundamental tool in image processing, including object detection in computer vision (Muthukrishnan & Radha, 2011). Because the MNIST dataset we used for our digit classifier uses fairly low resolution images, the Canny edge detector was a crucial addition, as applying edge detection to remove any extraneous features was critical in providing clean images for the classifier to work with.

**Algorithm Overview (Python)**

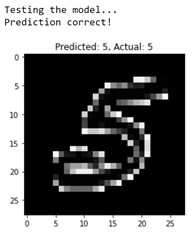
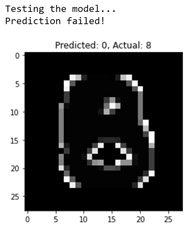
The Python portion of our code was completed using the scikit-learn library and the MNIST dataset. The scikit-learn library is a robust and well-documented machine learning library, and as such helped simplify much of the process of creating and training the SVM model. After weighing different options, we opted to employ a support-vector machine (SVM) model due to its simplicity and the nature of the application.

The MNIST dataset contains thousands of 28x28, 8-bit grayscale images of handwritten digits. Using this dataset, we were able to train our SVM model to recognize the digit being represented in the image. However, instead of using the MNIST dataset images directly from the official website, we found an alternative source that contains the same image data in the form of a csv file. Each row in the csv file constitutes one image, with the first column being the intended digit and the next 784 columns each having a value from 0-255, representing each pixel in the image. In this format, the SVM model was more efficiently able to evaluate the data.

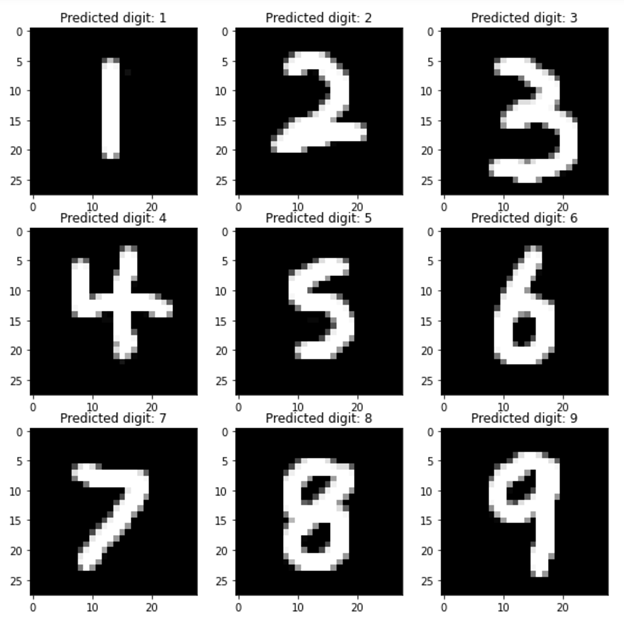
Objects from the Pipeline and GridSearchCV classes are what create and apply the SVM model. The Pipeline applies a series of transforms on the data (defined in steps) as well as an estimator to determine the fit. This Pipeline object is passed through the GridSearchCV object, which runs through a Python dictionary of different parameters to test and implements a score and predict function. However, because our digit classification model does not change significantly between different training runs, the GridSearchCV object outputs the same set of optimized parameters each time. By only supplying that one set of parameters and thus eliminating the need to test many different sets, we were able to cut down on computation time.

After the model was created and trained using the training dataset, we ran the testing dataset (1000 images in this case) through the aforementioned predict and score functions from the GridSearchCV class to apply the model to the testing dataset and calculate an accuracy score. Unsurprisingly, the larger the dataset used to train the model, the more accurate the model was in classifying digits, at the expense of increased computation time. We found that 8000 images to train with was the best option in terms of the tradeoff between training time and accuracy. With 8000 images, it took about three minutes to train and resulted in a 96.6% accuracy score after testing. 1200 images took just around 15 seconds and resulted in an accuracy score of 91.4%. Any training dataset larger than 8000 takes exponentially larger to train with minimal gains in accuracy.

After the training was completed, we included functionality in the code which outputs a randomly selected image from the MNIST dataset, its predicted value, and its actual value. Using this code, we were able to see that the trained model was able to accurately classify the images in the MNIST dataset, as indicated by the accuracy score.

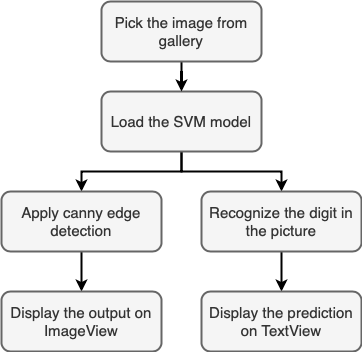
 

Pictured above are examples of both a correct and incorrect prediction. Discerning between 0, 3, and 8 was one of the most common mistakes the classifier made.   
 For further testing, we imported our own handwritten images. These were simply created in Microsoft Paint in the same format as the MNIST dataset (28x28, 8-bit grayscale). This was run through some code that would translate the image into arrays of values, just as how the data in the csv files were formatted. The same predict function was used to apply the model to these 9 digits.



With the model trained with 8000 images, our model was able to accurately predict what all of the digits were. However, with the 1200 image model, some of the common mistakes appeared, most notably with differentiating between 3 and 8.

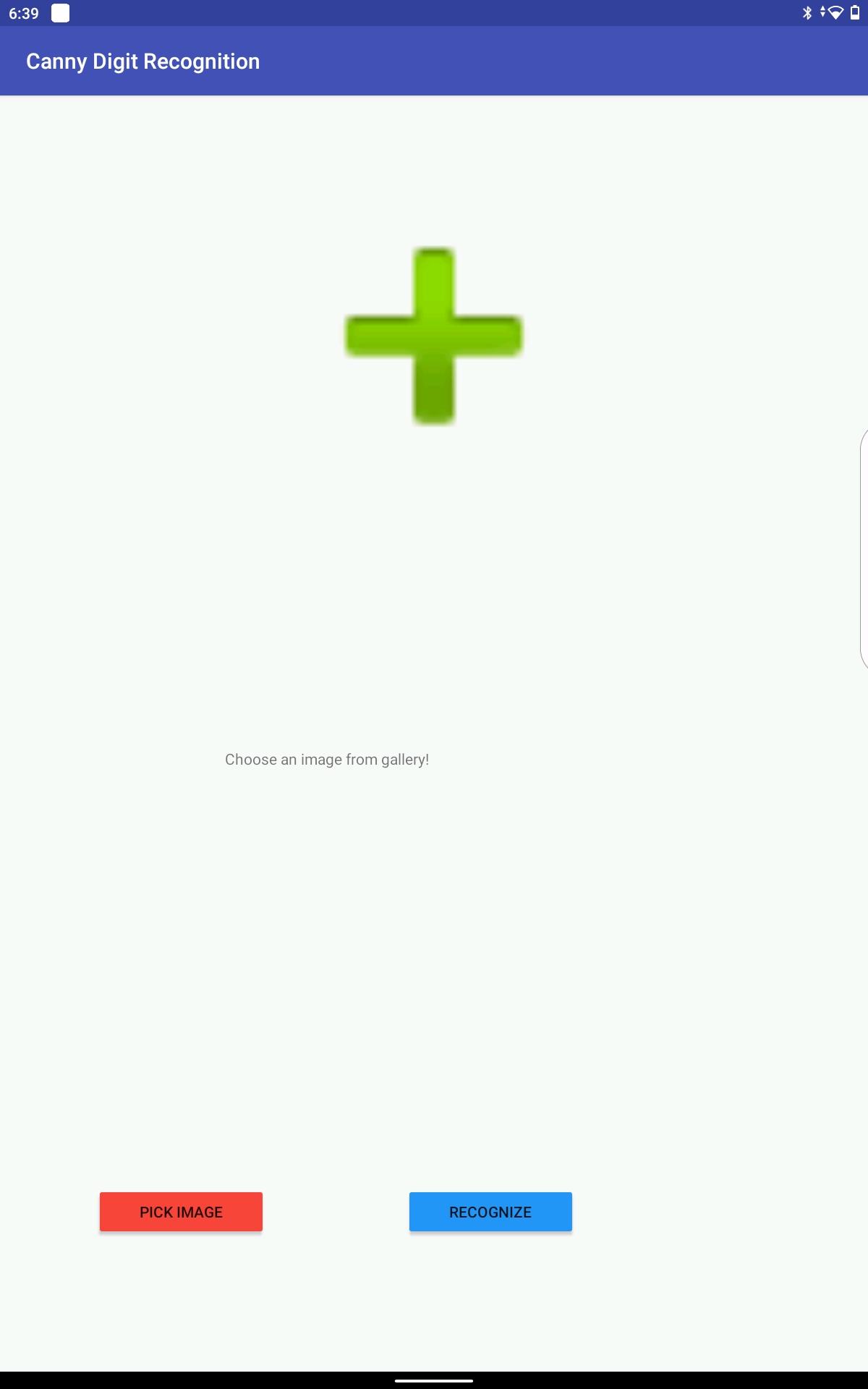
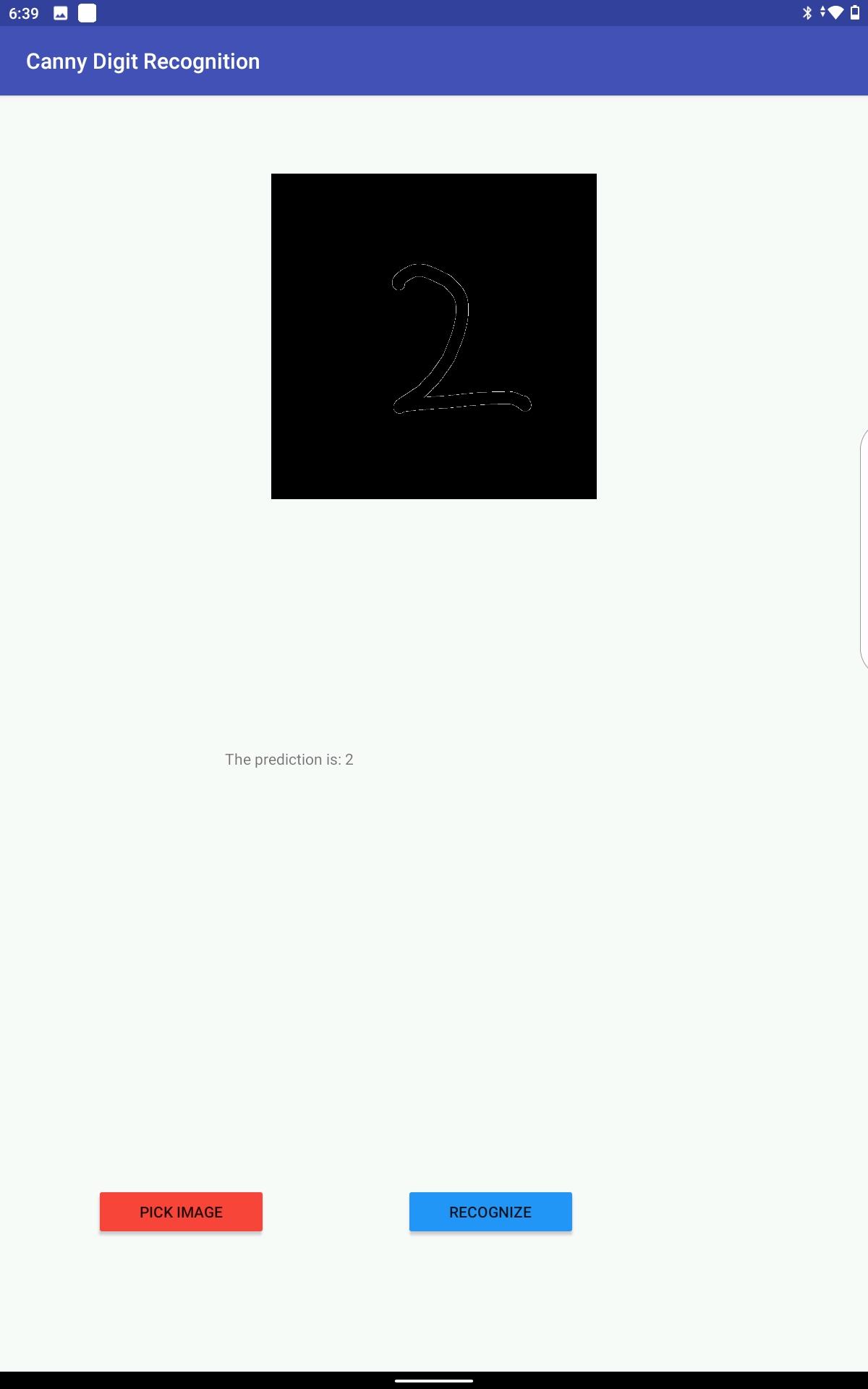
**Algorithm Overview (Android)**

****

As shown in the flowchart above, our Android app will first let users pick an image from the gallery. The target image that contains the handwritten digit must be stored in the device before opening our app. Once the image has been loaded successfully, the app will automatically perform canny edge detection and display the output on the screen. The canny edge detection is done by using the function from openCV directly. To prepare the image for edge detection, we first convert the input picture from png to bitmap. Then the bitmap is transformed into a matrix and converted from RGB to grayscale. This created matrix is used as the input for edge detection.

For the digit recognition, we imported a prepared classifier model in xml format, which was trained and saved using python. This saved model is put under the project directory and imported when the app turns on. When both the image and the classifier are successfully loaded, the recognition will start when the user presses the “Recognize” button. To begin the digit recognition, the input image in bitmap format is converted to matrix in grayscale. Then the matrix is resized to 28x28 in order to fit the requirement of the classifier. Finally the matrix is normalized and fed into the classifier. The prediction will finish instantly, and the result will be displayed on the screen.

**Overview of the User Interface**

****

The User interface is shown above, it contains two buttons, one for letting the user pick an image from the gallery, the other for starting the recognition. The text view in the middle is for showing users the instructions of using the app and the result of recognition. The image view will automatically display the output of edge detection after the user picks the image.

**Results**

After testing all of the ten digits, we find out that our classification is robust on digits 1, 2, 3, 5, 6, and 7. This result is worse than the performance on python. We think this is because the Android program manipulates the image data in a different way. When the program resizes the image to 28x28 and converts the bitmap to matrix, the shape of the digit changes and some crucial details for recognition might be lost during this process. Therefore making the classification less accurate.

In our proposal, we wanted to feed the output of edge detection to the classifier in order to increase the accuracy of recognition. However, we realized that this method is impracticable when testing the recognition. This is because our model is trained on the MNIST dataset, which contains images with black background and white text. Our result of edge detection is different from these pictures, so the classifier cannot correctly recognize the image after edge detection. As a result, we test our classifier model with images directly from the gallery and display the result of edge detection separately.

**Conclusion**

In conclusion, we were able to create a highly robust digit detector using an SVM model combined with the results of the Canny edge detector. Given our relative lack of experience with machine learning at the beginning of this project, we are highly satisfied with the results of the final product. Further work can be done by implementing our Canny edge detector to work with color images. We could also work on training and applying our classification model with higher resolution images.

**References**

Muthukrishnan, R., & Radha, M. (2011). *Edge detection techniques for image segmentation*. International Journal of Computer Science and Information Technology*.*

Sahir, S. (2019, January 27). *Canny edge detection step by step in python - computer vision*. Retrieved March 20, 2021, from https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123