Carson Crick and Ryan Williams

Professor Blanco

CSCI-B351

26 April 2020

K-NN Number and Symbol Recognition: Technical Paper

Our project goal was to create a simple calculator that takes handwritten numbers and symbols as inputs, classifies the inputs correctly most of the time, and outputs the result. We created a K-NN classifier to classify the handwritten characters. We wanted to make sure our classifier had plenty of uniform training data, so we decided to use the MNIST data set for our number testing and training sets. For the remaining testing and training sets, we had to create our own data because we couldn't find a third party data set that matches the MNIST data set format. Each of our images are 28 by 28 pixel grids containing grayscale values from 0 to 255. Our inputs were lists of images in the order of the equation from left to right. Our classifier would classify each character and then solve the equation using order of operations.

Our biggest challenge is choosing the balance between computation speed and classifier accuracy. For example, we can choose to have our classifier train on all 6,000 training pictures and 10,000 test examples in order for it to be very accurate but of course it would take much more time to compute. On the other hand, we can train it on a portion of the training and test examples to have it compute in less time but then it would be less accurate. Not only do we have to choose how much training and testing examples we want, but we also have to consider the k value. How large or small the k value is also factors into computation time and accuracy rate. Another challenge is the data we are using for our classifier. Our images need to be 28 by 28 and black and white images. We didn't want to scale down images because we didn't want to

lose pertinent information of the numbers/symbols. Another challenge is dealing with the fact that our classifier will never be 100% accurate. So in theory our calculator could output a wrong answer to your inputted mathematical formula because of misclassifications. Lastly, we had to create our own dataset of symbols such as +,-,* in order for our project to do mathematical calculations. This involved creating 28 x 28 grids of different variations of each symbol that we wanted it to be able to detect.

The most common variation you will see for this problem is using neural networks instead of a K-NN classifier. (Maybe go in depth about the way neural networks work for this problem.) With K-NN there are many different formulas to determine how close one picture resembles another. For example, Euclidean distance, Manhattan distance and our formula are all ways to obtain a “resemblance” value between two pictures. People can choose whether to maximize or minimize this value too. You would most definitely see a variation in the k values people use because each person’s algorithm performs differently given how they determine their “resemblance” values.

We are using two algorithms for this project. The first is a third party algorithm we are using finds the distance between two images by subtracting the grayscale values in the grids from each other and squaring the values. The second finds distance by getting the index values in both images from grayscale values that are above a preset whitespace threshold and finding the euclidean distance between the active points. Then we push the labels and image distances into an ItemQueue class. The ItemQueue class has two different ordered lists. These lists have the lowest distance values first and match the index of the distance with the corresponding label in the label’s list. Next, after the given image has the distance between it and the training data in the ItemQueue, we grab the top k distances with their corresponding labels. Finally, we find the plurality of labels in the top k labels and return that is the image’s classification.

Our classifier has two different time complexities depending on the distance formula that is being used for it. Our classifier with the third party distance formula has a time complexity of x*y*784 where x is the number of numbers and symbols the classifier is testing for and y is the number of training images for each number/symbol. Our classifier with our distance formula has a time complexity of x*y*(784 + p1*p2) where x is the number of numbers, symbols the classifier is testing for and y is the number of training images for each number/symbol, p1 is the number of points in the first image and p2 is the number of points in the second image. A point is defined from an item from the array that has a higher grayscale value than the whitespace.

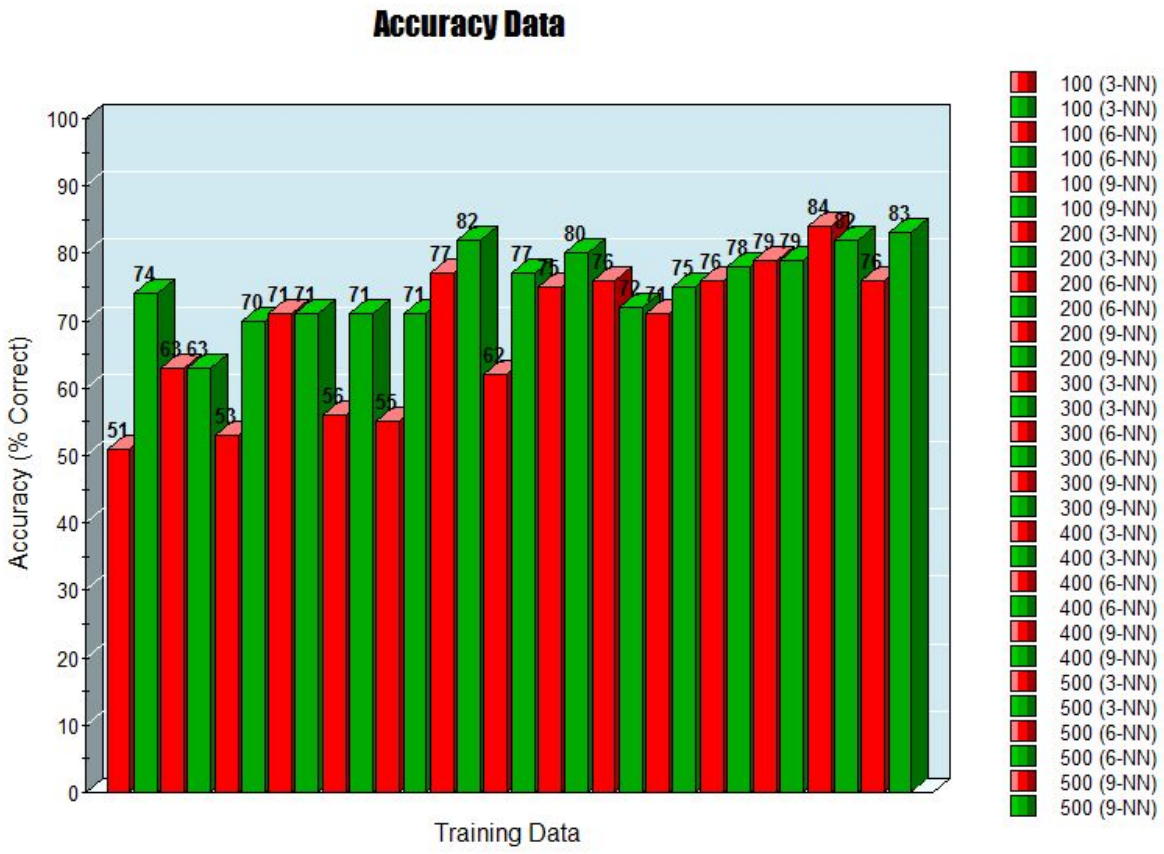
One of the limitations we faced is the MNIST dataset and our symbol dataset do not cover all handwriting types so our classifier is only accurate in the sense of the MNIST data and our symbol data. Additionally, our symbol dataset is not equal in size and in symbol variation to the MNIST dataset since we had to manually make our symbol dataset. Another limitation is we only deal with 28 by 28 pixel grids and no other sizes of images. Our classifier doesn't work with images greater or less than 784 pixels.

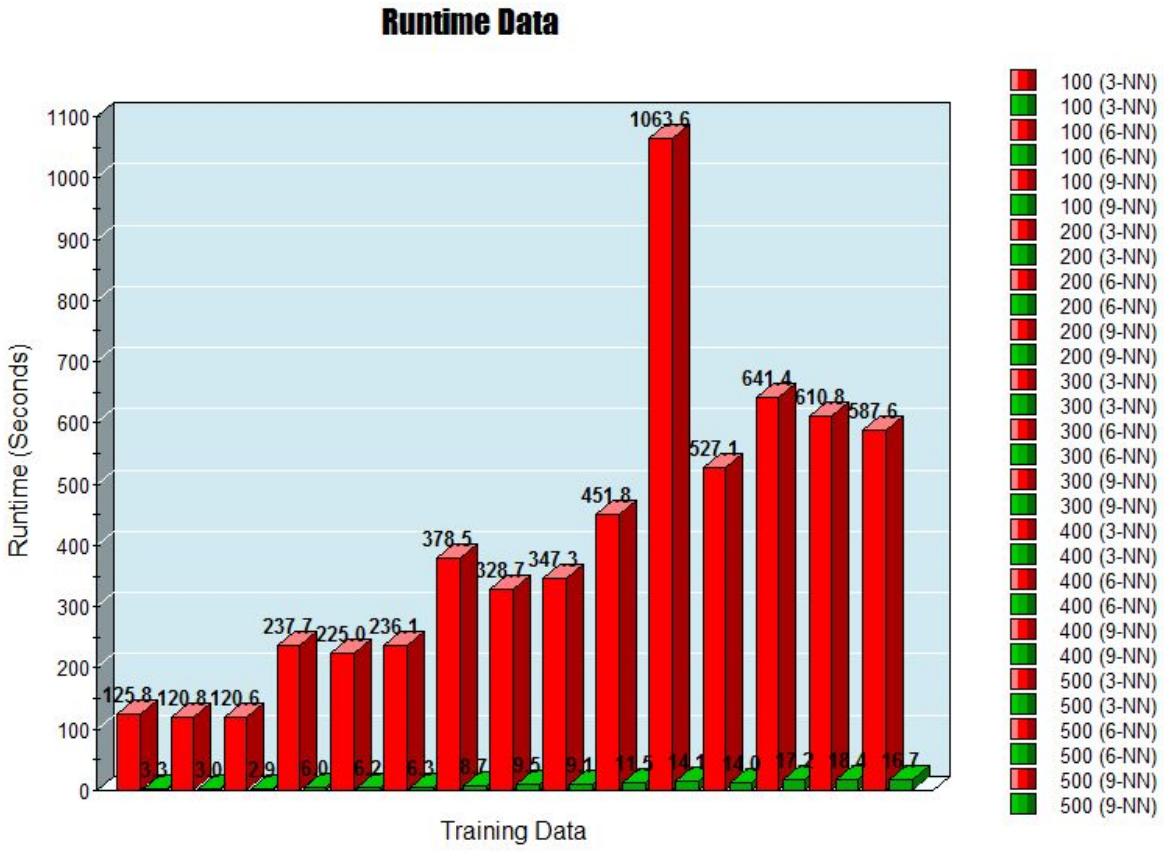
Some alternatives could be we could find a select amount of the best representations of each character and save it to use as training data. Right now we are grabbing the first x training images from the training set where x is a positive integer. By doing this, we could be able to use less and less training images and still have a higher accuracy value.

Describes how your solution models human thought processes.

Our program models a human thought process by using it's training data to classify the given image. This is similar to when a human is trying to identify a handwritten character. Our program goes back and looks at the numbers and symbols it has been given and knows it's classification and sees what it is the most similar to. Our program also models the human thought process by following order of operations when making calculations.

Our analysis shows the difference between accuracy and run time between the distance formula we created against the third party distance function we found. This will compare the two formulas using 3-NN, 6-NN and 9-NN with training data sets with 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 training images. These graphs are only showing the classifiers performance with recognizing digits 0-9 and excluding any symbols because symbols aren't included in the MNIST data set. We didn't make 100 training images for symbols so we decided to exclude them from our graphs.





We got these results by running our classifier with two different distance functions, one we created and the other is from a third party. In the graphs above, the red is with our distance function and the green is with the third party distance formula. Both functions were tested with 3-NN, 6-NN and 9-NN algorithms with training data from 100-500 going by 100 increments coming from the MNIST data set. Each test we ran accuracy and runtime tests for 100 random testing images from the MNIST data set. I expected the third party distance to be much faster, but not to the scale as seen in the graph. We do have one outlier with the data we collected. It was at 400 training images and 6-NN. I believe the time is so much greater because the laptop used to get the data was sleeping for a little portion of the run. The graph shows that the issues

with making distance formulas slower and complicated when dealing with larger amounts of data.

The third party distance formula by Priya Viswanathan determines the distance between two images by going through the image grid. It can be found on http://shyamalapriya.github.io/digit-recognition-using-k-nearest-neighbors/, and her github link can be found at https://github.com/anandsekar/datashakers-digitrecognizer/blob/master/python/classify.py. In short, the formula can be seen as a different form of Euclidean distance. It loops through the test and training picture's individual pixels, comparing their pixels' grayscale values (0-255) with each iteration. Therefore, instead of using pixel coordinates in the original Euclidean distance, we use pixel grayscale values.

Fundamentally, our program could have been improved if we took a different approach and used neural networks to classify the data instead of using a K-NN algorithm. This would solve our challenge of having to balance run time and accuracy because after the neural networks are trained, its weights are saved after the training is done. Therefore, we would be able to give the program as much training data as possible and it would affect the runtime of classifying the images.

We could also make improvements to our already existing algorithm. First, would be to step outside of the MNIST dataset and our symbol dataset to obtain more handwriting types and data in general. Second, would be to allow our algorithm to detect and complete more complex mathematical functions such as logarithmic functions, exponential functions, solve equations of the line, solve systems of equations, etc. Lastly and more complexly, would be to allow a user to take a picture of their handwritten mathematical equation and upload it for our algorithm to solve.

One of the lessons we have learned is the importance of efficiency when dealing with larger amounts of data. In our program, we would only have to deal with around 10,000 training images and having a slow algorithm really increases the time it takes to create an output. Using the third party distance formula as described above allowed big improvement to our program because it allowed us to give the classifier more training data and have the classifier complete its function in a reasonable amount of time.