Carson Crick and Ryan Williams

Professor Blanco

CSCI-B351

5 May 2020

K-NN Number and Symbol Recognition: Technical Paper

Problem Space

Our project goal was to create a simple calculator that takes handwritten numbers (0-9) and symbols (+, -, *, /) as inputs, recognizes them, and outputs the result. To do this 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. The MNIST dataset provides 10,000 training images and 6,000 testing images. Each image was a handwritten digit centered in a 28x28 image, each point in the image was a grayscale value (0-255). For our symbol data, we had to create our own data because we couldn't find a third party data set that matches the MNIST data set format. Our symbol data was made to resemble the MNIST data format so that all of our data could be uniform. Our inputs were lists of images in the order of the equation from left to right. Our classifier classifies each character and then solves 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 60,000 training pictures and 10,000 test examples from the MNIST dataset in order for it to be very accurate but 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 also factors into computation time and accuracy rate. 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 (+, -, *, /) in order for our project to do mathematical calculations. In order to keep our data uniform with the MNIST data format we had to manually create our symbols in Python using 28 x 28 grids and grayscale values. Because we had to create them manually it was incredibly time consuming. We ended up creating a total of 41 training images and 31 testing images.

The most common variation you will see for this problem is using neural networks instead of a K-NN classifier. 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.

Our Algorithm

We use two different formulas in our K-NN algorithm to determine how much one picture resembles another. The first is a third party formula which finds the distance between two images by looping through their grayscale values. With each iteration the formula obtains a grayscale value of the same index from the training and testing image. The formula then subtracts these two grayscale values and squares them. The final result is a sum of these differences. 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 valid points. Then, we push the training images' labels with their corresponding distances from the test image into an ItemQueue class. The ItemQueue class has two different ordered lists. These lists have the lowest distance values first. Next, after the given testing 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 most prevalent label 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 variation to the MNIST dataset since we had to manually make our symbol dataset. Lastly, because of our use of the MNIST dataset we were limited to only dealing 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.

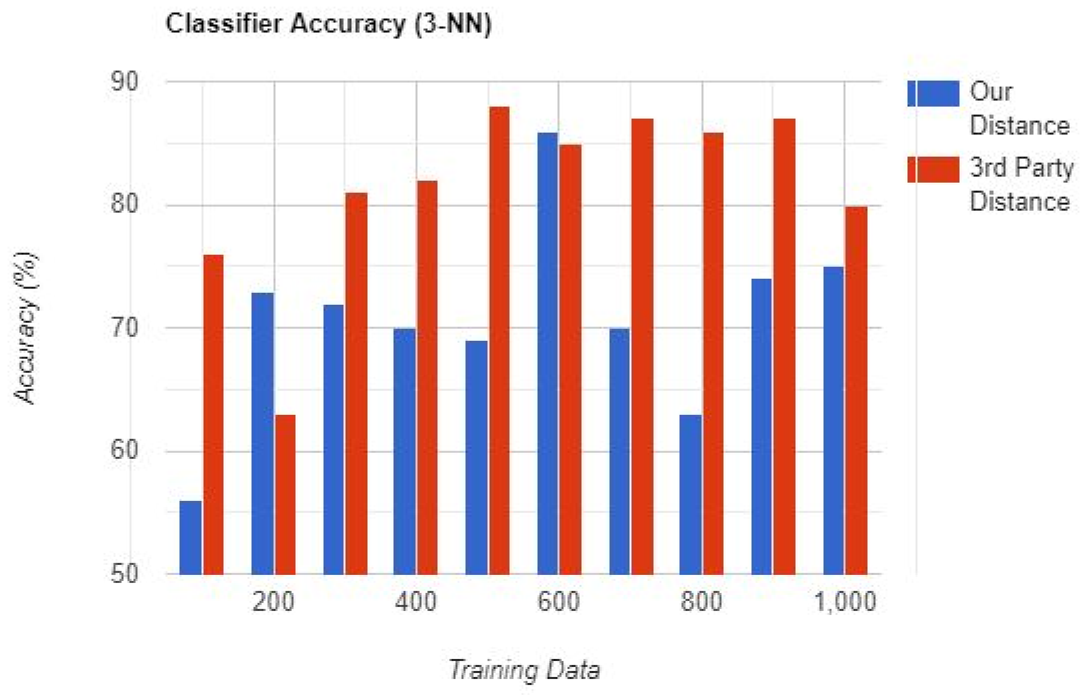
Some alternatives could be we could find a select amount of the best representations of each character and save it to use as training data. Our current process is to obtain the first x training images from the training set where x is a positive integer. By doing this, we could be able to use less and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

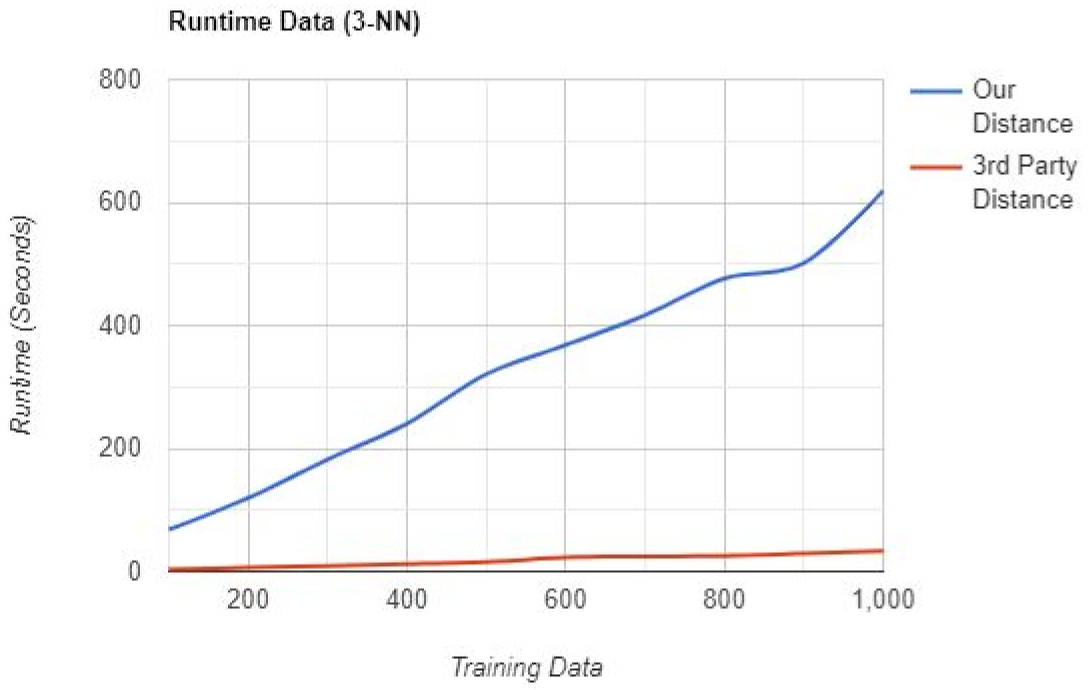
Relation to Human Thought Process

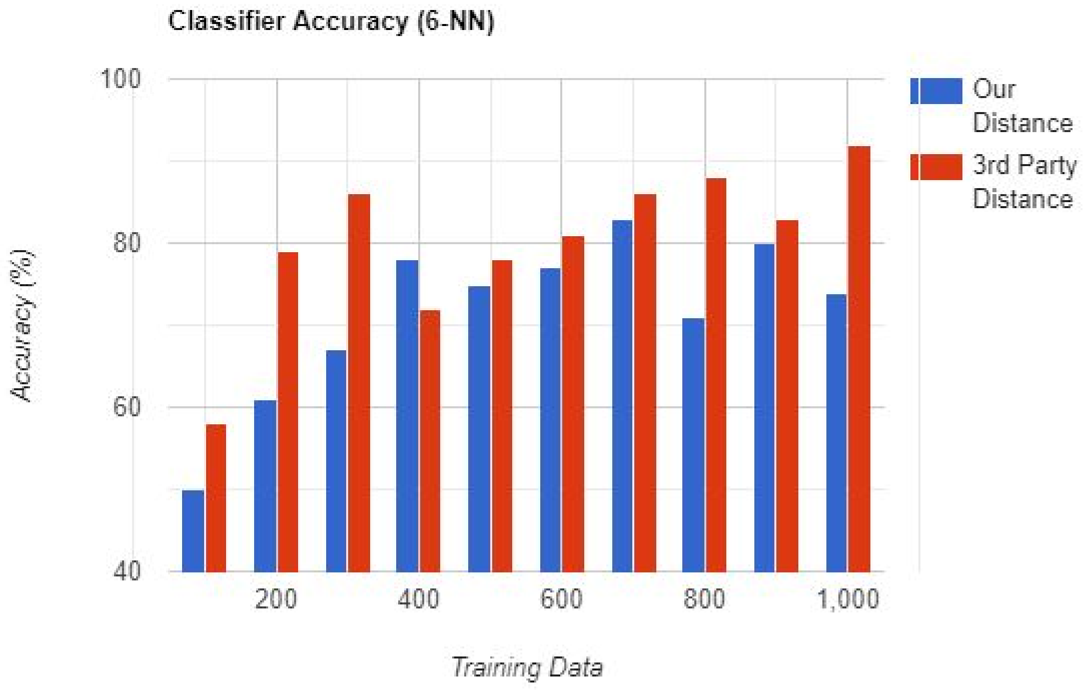
Our program models a human thought process by using it's training data to classify the given image. This is similar to when a human is trying to identify a handwritten character by using previous experience and memory of what they have learned. Our program goes back to look at the numbers and symbols it has been given (similar to human memory), looks at their labels, and sees what it is the most similar to. Whatever the number or symbol the test image is most similar to is it's classification. Our program also models the human thought process by following order of operations when making calculations.

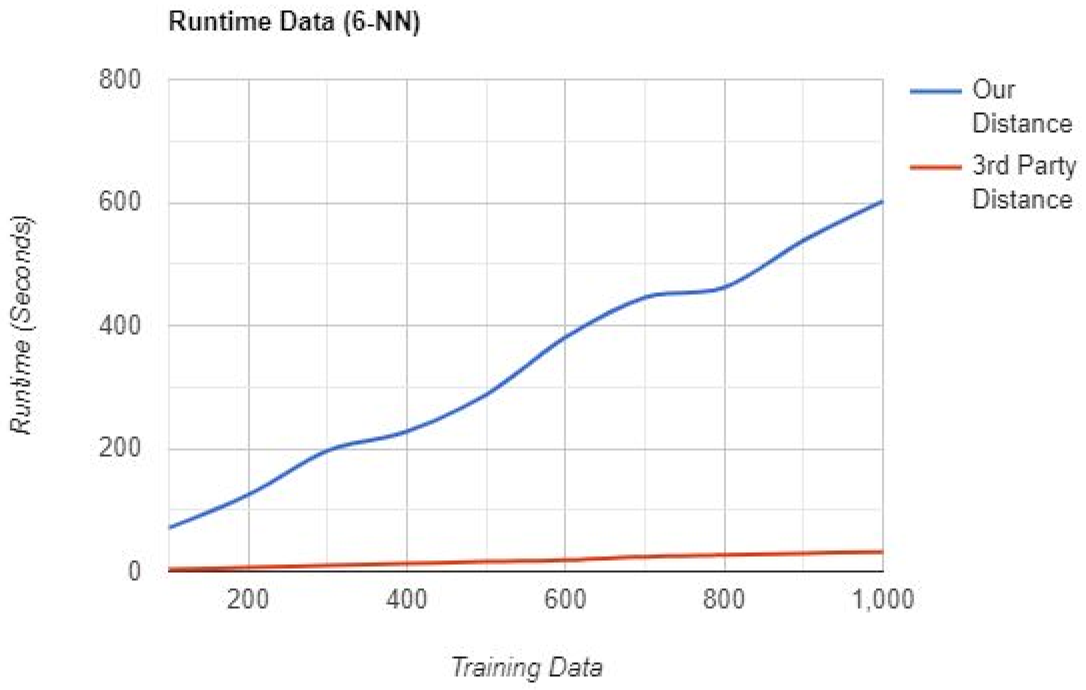
Analysis and Performance

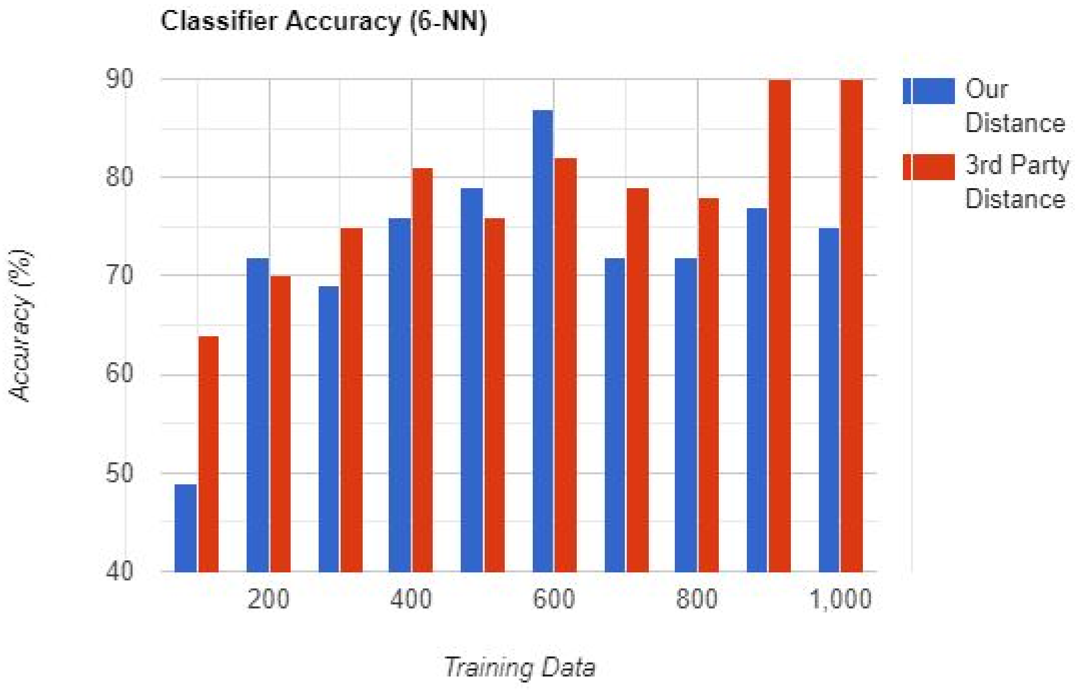
Our analysis shows the difference between accuracy and run time between the distance formula we created and the third party distance function we found. We compare the two formulas using 3-NN, 6-NN and 9-NN with training data sets with 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 training images. These graphs only show the classifiers performance with recognizing digits 0-9 and exclude any symbols. For each test, we used 100 random testing images from the MNIST data set, again, excluding symbols. We decided to exclude symbols because of our small amount of training and testing images for them.

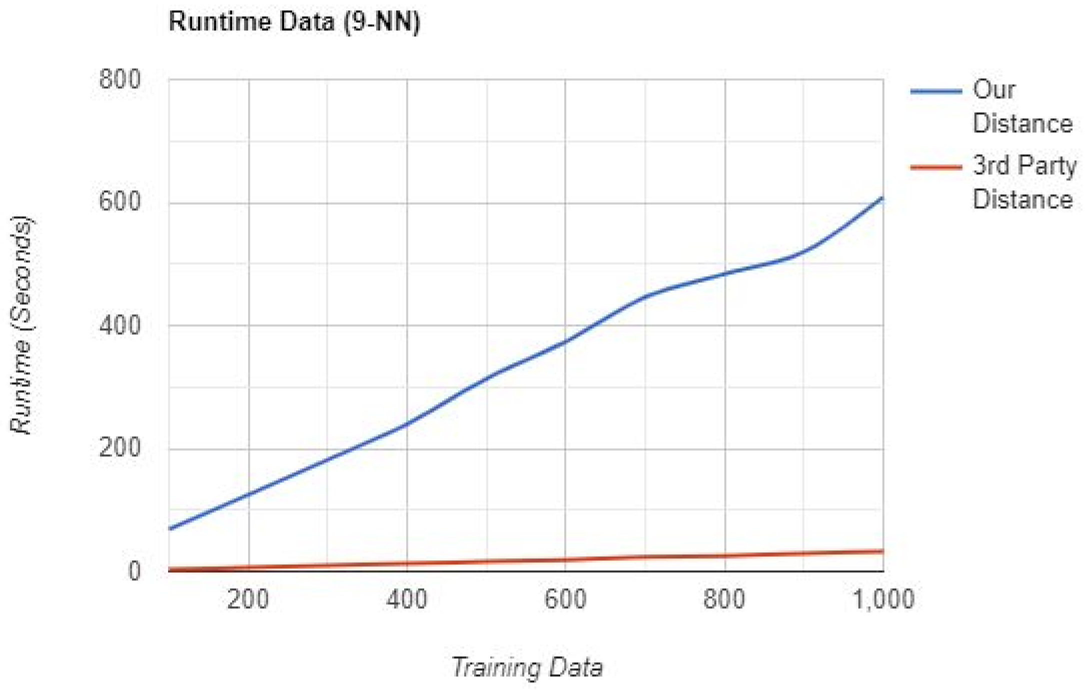

Classifier Accuracy (3-NN)
Accuracy (%)
90
80
70
60
50
200
400
600
800
1,000
Training Data
Our Distance
3rd Party Distance


Runtime Data (3-NN)
800
600
400
200
0
Runtime (Seconds)
200
400
600
800
1,000
Training Data
Our Distance
3rd Party Distance


Classifier Accuracy (6-NN)
100
80
60
40
Accuracy (%)
200
400
600
800
1,000
Training Data
Our Distance
3rd Party Distance


Runtime Data (6-NN)
Runtime (Seconds)
800
600
400
200
0
200
400
600
800
1,000
Training Data
Our Distance
3rd Party Distance


Classifier Accuracy (6-NN)
Accuracy (%)
90
80
70
60
50
40
200
400
600
800
1,000
Training Data
Our Distance
3rd Party Distance



From the graphs, it is very clear that the 3rd party formula completely beat our formula both in accuracy and in runtime. For accuracy, the 3rd party is more accurate overall because of its use of grayscale values. It takes into account all points whereas our formula only takes into account points that reach above a whitespace threshold. Our formula treats all valid points the same, even if one point has a grayscale value of 201 and another of 255. The 3rd party formula, however, does not. It compares the training picture and testing picture pixel to pixel, taking into account each grayscale value difference. Though it is important to note that in some situations our formula was more accurate. When it comes to runtime, it is obvious why our formula takes much more time than the 3rd party. Our formula has to first find the valid points and then has to compute Euclidean Distance multiple times to find the shortest distance to each valid point. However, the 3rd party formula just obtains grayscale values and puts them in an equation for each pixel. In general, our formula does much more computations. Therefore, the 3rd party formula is more optimal to use. Concluding from our results, it is best to use a k-value of 3 and to read in 600 training pictures. When using this, both formulas flaunt an accuracy rate of ~85% and compute in a relatively short amount of time (especially the 3rd party formula). With K-NN, a higher k-value does not necessarily it becomes more accurate. A k-value of 3 outputs a high accuracy rate given enough training data, therefore a higher k-value is not needed. A choice of 600 training points is also optimal because it offers a high accuracy with all k-value tests without the price of a long runtime such as with 1,000 training points.

Outside Code

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.

Improvements

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 without affecting 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. With our formula, we would have the computation time to read in x amount of training images plus the slow computation time of our formula classifying. 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.