Using KNN to Recognize Handwritten Numbers



**Daymon Wu**

daymon.wu90@qmail.cuny.edu

# Abstract

Recognizing digits with computers is interesting in the area of machine learning. This project’s goal is to use the K-Nearest Neighbors algorithm to identify different handwritten numbers. It is a job that can be used in everyday applications like digital document processing and filling forms automatically. The main idea was on fixing the KNN algorithm by adjusting the number of nearest neighbors k, an important parameter that affects the algorithm’s accuracy.

The method involved preparing a dataset of digit images by standardizing and flattening them to create consistent features. The KNN classifier was applied with a goal to find the best k value ranging from 1 to 11 through testing. This range was chosen because tests that gave higher values often led to over-generalization, which is bad for accuracy.

Performance was evaluated using metrics such as overall accuracy and per-class accuracy, that we got through a confusion matrix. Results showed that the best k, from our tests, enhanced the classifier's accuracy compared to using random values. The project concluded that while KNN is simple, its effectiveness depends on tuned settings and careful data handling. Future work could explore the use of more advanced image processing techniques to further improve accuracy and efficiency in real world tasks.

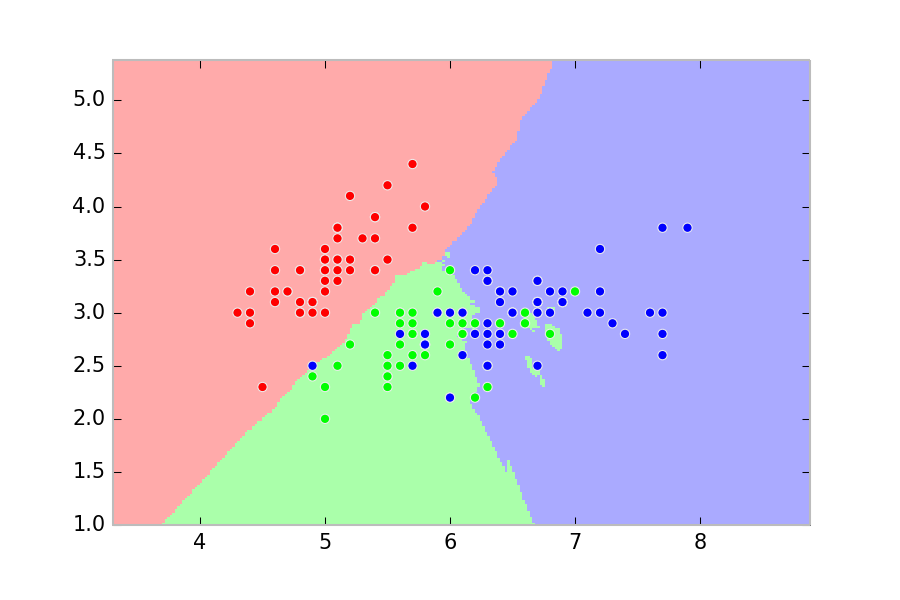
# Introduction

Machine learning plays a big part in our future, it offers tools that can learn from data to recognize numbers written by different people in different styles. The K-Nearest Neighbors algorithm is a classifier that is a powerful method used widely in machine learning for classifying objects based on their closest neighbors in data. For example, if you want to identify a 2 written by a person, KNN can look at examples that are similar to the 2 you are trying to recognize. But KNN comes with its own challenges, it can decide which neighbors are the most important.

This decision is based on a setting called k, which is the number of nearest neighbors the algorithm considers when it makes its classification. The choice of k is important to how well KNN can perform this task. If there are too little neighbors, the algorithm might miss small details leading to wrong guesses. Too many, and it might overlook the unique details of the number it's looking at. In this project, my goal is to find the best k that would allow the KNN algorithm to accurately classify handwritten digits from a set of images. I cleaned the images by making sure they were consistent in size so that the algorithm could compare them effectively.

Then I tested different values of k to see which provided the best balance between being too sensitive or too general. My goal in this paper is to show how changing parameters in machine learning tools like KNN can improve their ability to make sense of handwritten data, making these tools more useful for real life applications.

# Methodology



The image displays a two-dimensional scatter plot used to depict the classification boundaries generated by a K Nearest Neighbors algorithm. Each point on the plot represents a data sample, and the color of the point indicates its category.

The background color shows the decision regions determined by the KNN classifier. These regions show where new data points would likely be classified based on the majority vote of their nearest neighbors.

This visualization highlights how KNN looks at the 'nearest neighbors' essentially the closest points to decide the category of new, unlabeled points. The decision boundaries, where the colors change, show how the classifier divides up the space based on the data it has seen. This plot not only helps in understanding the distribution of data but also in observing how well the KNN algorithm can segment the data into different classes based on similarity in features.

Now instead of just 3 regions and colors we need 10 for digits 0-9.

## Preparing data

To do this we need to make sure all the handwritten digits, which are just pictures of numbers, look consistent to the computer. Each image was adjusted to be the same size and then transformed into a list of 256 numbers that represent each pixel's brightness. Then we adjusted these numbers so that they all had a similar range, helping the computer to not favor one picture over another just because it was brighter or dimmer.

## Implementing KNN

To classify a new digit, the KNN algorithm looks at all the known digit images and measures how close they are to the new one. This can be done by finding the straight line distance between two points on a map. It can pick the closest images, the number of which is defined by k, that is our new key setting.

## Optimizing K

To choose the right k we have to be very careful. If there are not enough neighbors we could make the decision too sensitive to different images, while too many can impact the decision because it has too much input. I used different k values from 1 to 11 to see which worked best. I used a fair method where the dataset was split into five parts. Each part was a test while training the algorithm on the others. This can help make sure our solutions are solid and would work well on any new data.

## Understanding results

To understand how well my solution was working, I looked at two things. The overall accuracy, it tells us how often the algorithm guessed the digit correctly and the confusion matrix which is a table that shows which digits are confused with others. These tools helped me see not just if the algorithm works, but where it struggles. My goal was to test the KNN algorithm's ability to recognize digits in a way that mimics real-world tasks. By tuning the k value and testing the algorithm, we can understand how simple changes can improve performance.

# Implementation & Experimentation

## Description of the dataset

The dataset used in this project includes images of handwritten digits, same to those you might find in everyday items like mail or forms. Each image is made to fit a size of 16x16 pixels, which turns each image into a list of 256 grayscale values. I split these images into two groups: one for training our model and the other for testing how well it works. The dataset is to cover a wide range of handwriting by different people including 7,291 training images and 2,007 testing images.

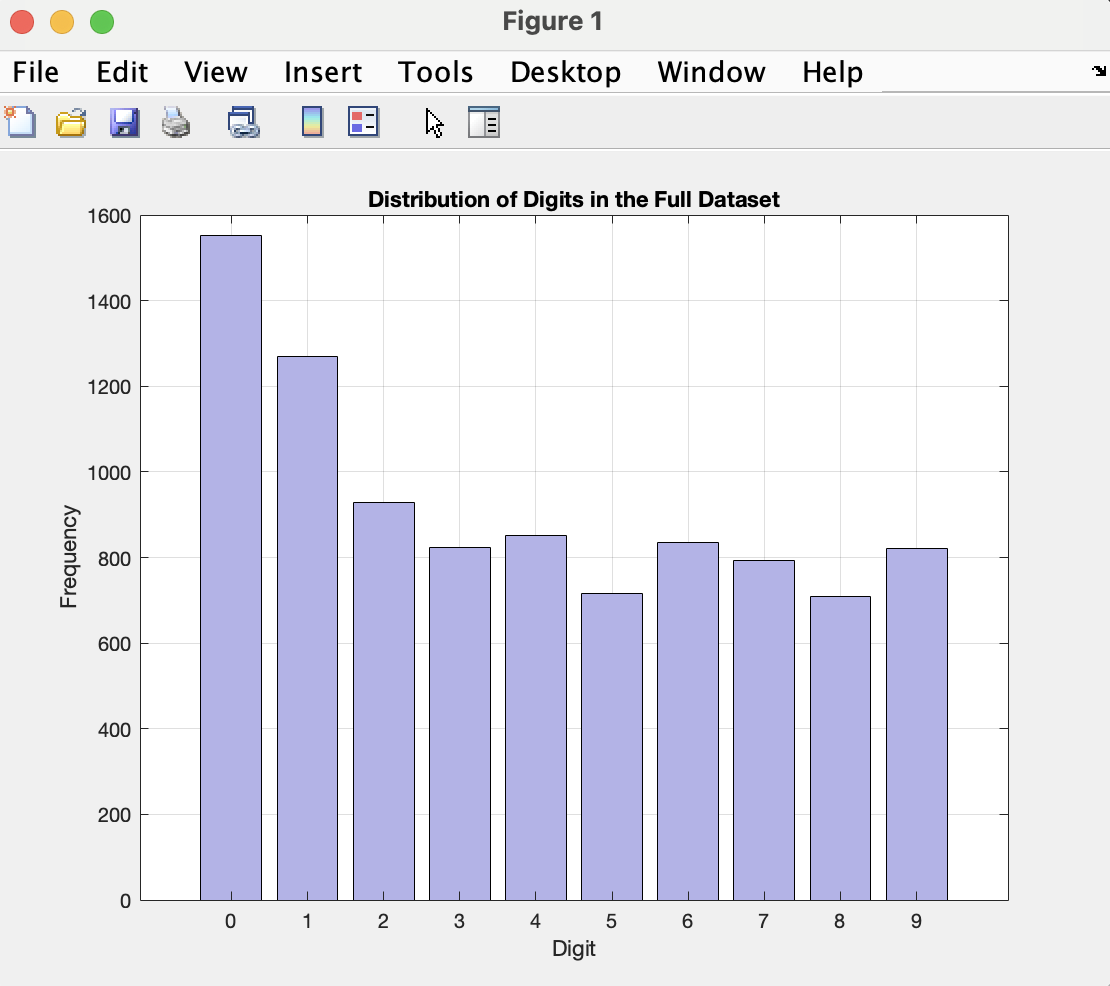


Figure 2: This is a graph of the digits in the dataset

**3.2.1 Loading Data**

To modify the KNN algorithm I used Matlab. To input the data so we can modify the KNN algorithm I used the built in loadData function because it can read dataset from text files directly into the Matlab workspace. I also used the readMatrix function so I can quickly load the data sets so it can be used in machine learning tasks.

**3.2.2 Extraction**

After the data is loaded I need to process the images to a format that we can use in our next steps. I used the flattenImages function to convert each image into a 256 element vector because the size for each image is 16x16. Flattening the image is super important because it can better calculate the accuracy computations. Standardizing input data and organizing features for distance calculations is important in KNN.

**3.2.3 KNN Classifier**

After the data is ready, knnClassifier computes the distances in the test and training areas to identify the k nearest neighbor. The only way to implement how KNN makes decisions and classify new digits is to base it on learned examples in the training and testing files.

**3.2.4 Optimizing of K**

To dynamically find the most effective number of neighbors that balances. The optimizeKValue function is to determine what is the best k. This function tests k in a loop iteratively. It keeps incrementing until it finds one that has the best accuracy on the test data.

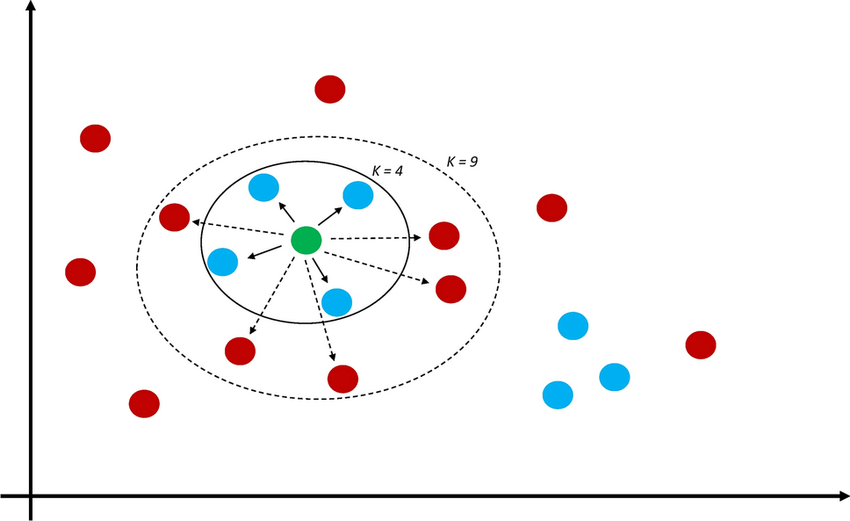
**3.2.5 How does k play a role**

K in KNN stands for the amount of similar photos it decides to look at to help guess the object or number in that image.

If the k is small like 1 or 2 it means it can only look at 1 or 2 images to compare to the new image. This can be good if those 1 or 2 images are very similar to each other but in our case we have 10 digits and they look different from each other. Guesses can be wrong if the images are too weird or different in a way you do not expect.

If the k is big like 10 or more there are a lot more images to help make your guess. If you have that many images you get a better idea of what to expect and usually helps avoid getting tricked by the weird images. But if there are too many images it might take longer and there might be images that aren't very similar at all and that can also mess up the guesses.

So in the function yPred, in terms of distance calculation it measured how far each known image is from the new image. In terms of sorting, it sorts all the known images by how close they are to the new one. In terms of selecting neighbors, it picks the closest k image. And in terms of which k it should pick, the computer looks at what numbers are on those k images and chooses the number that appears the most often.



This picture helps explain why the number of neighbors we look at matters. Looking at a few neighbors might not give enough info and can be confusing, while looking at many might include points that aren't very similar, which can also lead to wrong guesses. This shows the balance needed in picking the right number of neighbors to get the best guess.

**3.3**  **Description of performance measurement, confusion matrix, per-class accuracy, and overall accuracy**

**3.3.1 Performance**

To measure the performance, I used a few methods. The Overall Accuracy which is the percentage of guesses guessed right out of all the guesses. The Confusion Matrix that shows which numbers are getting mixed up. And the Accuracy for each individual number.

**3.3.2 Validation**

I tested how many neighbors the tool should think it should make its best guess by trying different numbers from 1 to 11. This process involves splitting the training group into smaller parts and making sure our findings are consistent across each part. The idea was to find a sweet spot where the tool is just right.

**3.3.3 Results**

Figure 1:

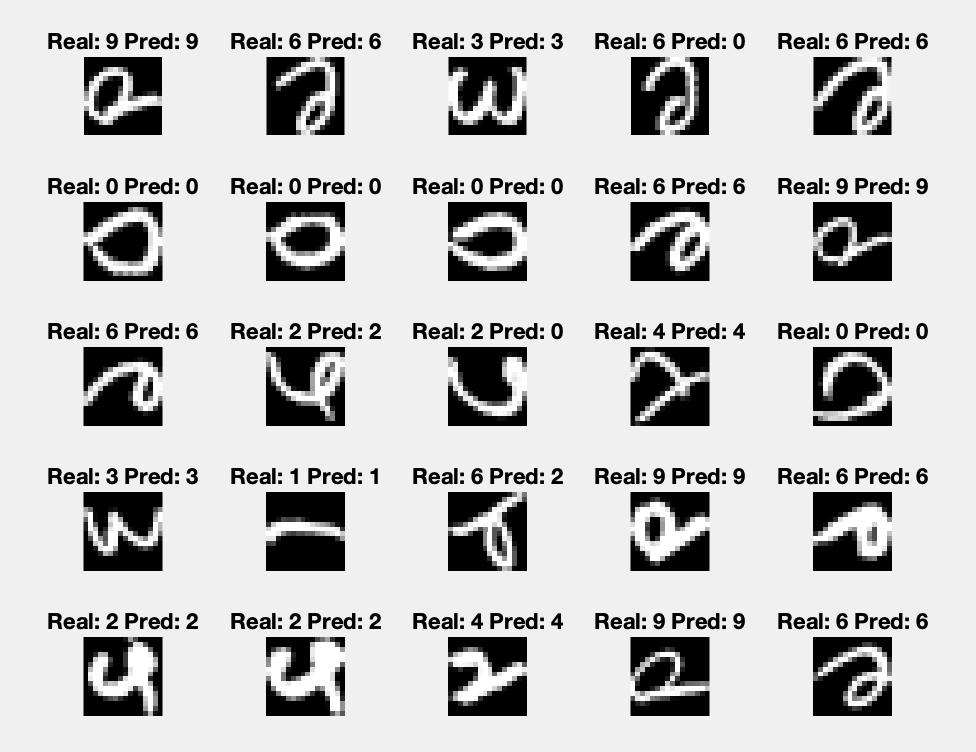


Figure 1: This figure shows a selection of handwritten digit predictions made by the modified K Nearest Neighbors algorithm. Each image panel shows a digit as interpreted by the algorithm, alongside labels indicating the actual digit and the digit predicted by the model.

Table 1: Confusion Matrix

353 0 3 0 0 1 0 1 0 1

0 255 0 0 6 0 2 1 0 0

6 0 176 1 0 2 2 5 6 0

2 0 4 150 0 7 0 1 0 2

1 3 4 0 177 1 1 3 0 10

4 1 1 4 0 139 1 3 6 1

5 0 3 0 1 2 159 0 0 0

0 0 1 1 4 0 0 139 1 1

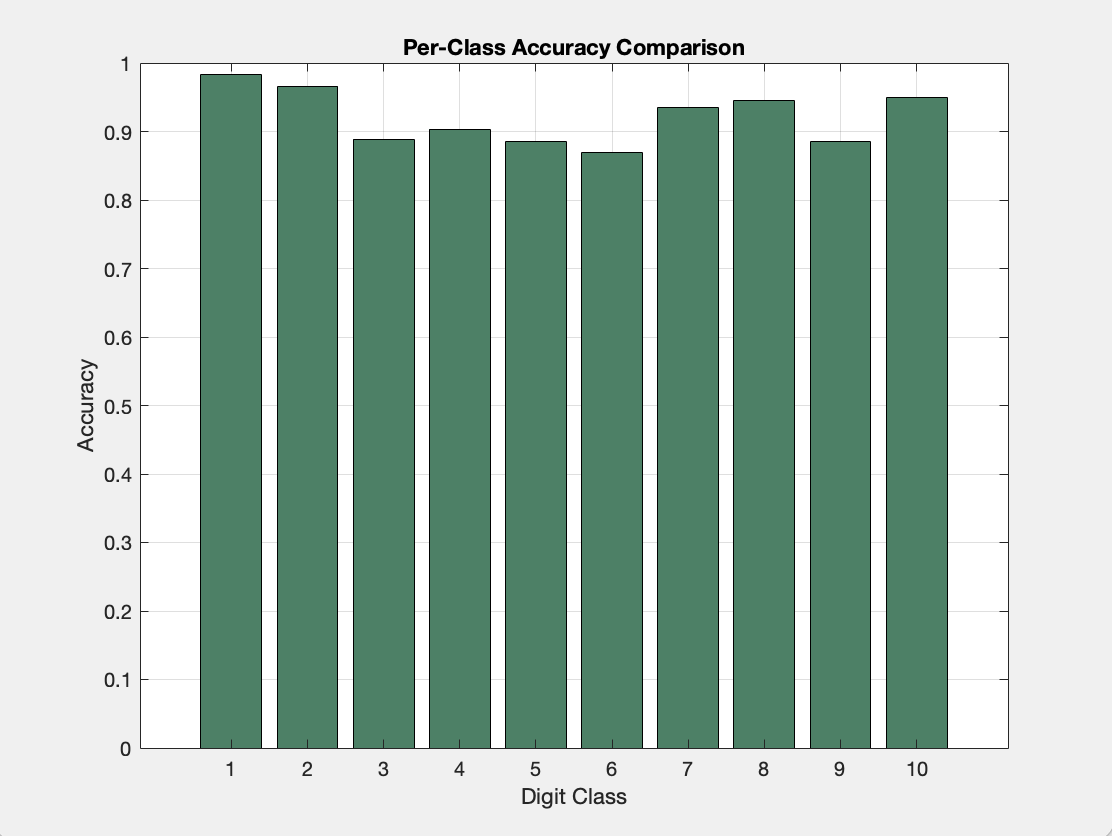
5 0 3 4 0 2 0 3 147 2

1 0 0 0 2 0 0 5 1 168

Figure 1: This is a table used to evaluate the accuracy of a classification model. It shows how well a computer program, like one designed to recognize handwritten digits, identifies the correct number from 0 to 9. Each row in the matrix represents the actual number and each column shows the number the program thought it saw.

This matrix helps us see not just whether the model is generally good or bad, but specifically which numbers it gets confused with. This can help in improving the model, like training it further where it's weak or adjusting how it looks at the images.

Table 2: Accuracy for Each Number



Digit 0 | 98.33%

Digit 1 | 96.59%

Digit 2 | 88.89%

Digit 3 | 90.36%

Digit 4 | 88.50%

Digit 5 | 86.88%

Digit 6 | 93.53%

Digit 7 | 94.56%

Digit 8 | 88.55%

Digit 9 | 94.92%

Figure 2: This is a bar chart that displays the accuracy of recognizing numbers from 0-9, this is from data from the confusion matrix.

The overall accuracy is 0.92825.

# Conclusion

In this project, I made improvements to a basic machine learning method called K Nearest Neighbors to help it recognize handwritten numbers better. By tweaking some settings and using smarter ways to process images, our modified version is more accurate and handles different handwriting styles well. The results show that a few thoughtful changes can make a big difference in how well the system works. Moving forward, we can plan to make it even faster and see if it can be used for other types of image recognition tasks.

# References

1. <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
2. <https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn>
3. <https://scikit-learn.org/stable/modules/neighbors.html>
4. <https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/>
5. <https://towardsdatascience.com/knn-algorithm-what-when-why-how-41405c16c36f>
6. <https://www.youtube.com/watch?v=Bb5VmXRtHsc>
7. <https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/>
8. <https://www.youtube.com/watch?v=HVXime0nQeI>
9. https://www.researchgate.net/figure/k-nearest-neighbors-A-diagram-showing-an-example-of-the-k-nearest-neighbor-machine\_fig1\_356781515