Handwritten Digit Recognition Using KNN and Kmeans

Authors: Wendy Woodin, Essam El Messiri, and Christi Balaki
Link to demo video: https://www.youtube.com/watch?v=HFPJ8wXrpnk&feature=youtu.be

Overview

We implemented a machine learning-based program capable of handwritten digit recognition. Our implementation employs two different algorithms to achieve this: kNN (k-Nearest Neighbors) and k-means clustering, an algorithm we used as an extension to our project. We evaluate the two methods and compare the accuracy achieved when different values of k are used.

Planning

As detailed in our initial draft specification (here) and final specification (here), our planned core functionality included reading in, parsing, and preparing the handwritten digit images and data from the MNIST database, implementing the KNN algorithm (using a Euclidean distance function), and evaluating the accuracy of our algorithm (including a comparison of the success rates for different values of k). Two of the potential extensions we planned to consider after completing the core functionality of our implementation included an exploration of other distance algorithms such as Kmeans and the creation of an interactive feature allowing the user to run different algorithms with different optimizations to compare their relative accuracy.

In implementing our project we succeeded in following our specification and planned schedule. Wendy completed the first part of the aforementioned plan (retrieving the data, parsing it and preparing it for processing), Essam implemented the kNN algorithm (with help from Wendy), and (as an extension) Christi (also with help from Wendy) completed the k-means clustering algorithm. Fortunately, we did manage to complete two of the planned extension exercises: (1) We have a user-friendly command-line system that makes it easy to try the two algorithms in different combinations, using different values for k, and visualize the results of those algorithms for easy interpretation in a graphical form and (2) we implemented a second algorithm, namely the k-means clustering algorithm.

Design and Implementation

We implemented the functionality in different files using the class system for Python. This was helpful for dividing the functionality so we could work on different parts at the same time and make sure that we could switch out implementations. We had some difficulty with the interaction of the appliance with Python packages (after some debugging of this, some of us moved outside of the appliance due to the complications of multiple Pythons being present on the appliance and causing pip install issues—this also caused a move away from code.seas to GitHub for better file sharing).

We tried to keep functions separate within the classes as well. For instance, at first in main.py, the visualization of the user's input was embedded within the if statements of the prediction runner. By separating the prediction running and the visualization, we were able to

condense a lot of similar code and reduce the number of functions needed in the visualization class from six to three.

Implementing user interaction was difficult at first. We wanted a way for the user to be able to choose algorithms and k-values for those algorithms without having to expose too much of the underlying implementation. After a brief sojourn into regular expressions, we instead used command line. This allowed us to prompt the user if they had problems with how to input their choices or inputted invalid choices

Our implementation of KNN had a couple of bugs along the way. Firstly, we had to figure out how to isolate the test points and be sure that we were not testing a point against itself. We accomplished this by using indices and the random library in Python so we could randomly assign points to be tested. This made the testing more fair, but also more difficult because the accuracies can change on any given run of the system. We switched out the original euclidean distance calculation for one of the numpy library, which marginally reduced the runtime.

The implementation of Kmeans also had many bugs in its original implementation. First and foremost, we had issues on how to define the clusters in relation to the multiple centroids and how to keep track of them. First, we attempted to use a dictionary to store the clusters using a key, value pair in which the key would contain the index of the closest centroid in the centroid list and the value would contain the points that were assigned to that centroid, or the clusters. Unfortunately, this resulted in poor label manipulation and it proved too difficult to test the various point's labels using this method. In addition, the dataset was very large, and it was difficult to check the code's functionality as the code is NP-Hard to optimize. To resolve this problem, we had a different approach in which the point that defines an image is no longer a matrix, but a vector. The images are flattened in order to use the vector implementation accurately. In order to easily access the labels for both the clusters and centroids, both were defined by indices. Although this implementation of Kmeans is successful, it takes a quite a bit of time to run the implementation because it must parse through a fairly large dataset. Using a smaller k value decreases the running time; however, it also decreases the accuracy of the algorithm.

This long running time made testing the Kmeans algorithm for accuracy and time much more difficult, especially since we were using a random testing and training set with random centroid seeds so some cases, such as centroids that have no points associated, only cropped up occasionally for debugging. We also had the problem of figuring out how to assign labels to the centroids. We ended up finding the most common label in the cluster by looking up the vectors in the dataset and using the indices to find the label, but to get this to run in a reasonable time we had to shorten the length of the clusters. Our accuracy is also variable, ranging from as low as 6% to as high as 35%, which could be due to the changes such as shortening the clusters that we made so the code would run in a reasonable time.

Reflection

We were pleasantly surprised by the helpfulness of using command-line inputs. It reduced the number of cases we needed to check because the command line is very good at restricting user inputs. It also made prompting the user for input much easier.

We were not-so-pleasantly surprised by our accuracy levels for KNN. We were not able to get the accuracy up to the desired 80%. We tried optimizations such as weighting the votes, but most of these actually seemed to cause the accuracy to go down. To investigate this, we wrote a separate file (under report in our code) that analyzed the distances between digits that are the same and digits that are different but similar. We found that while the means of all of these distances are different, the distributions are partially overlapping, which could account for some of the problems we had with improving the accuracy.

We were also not-so-pleasantly surprised by how slowly KNN runs. To try to address this Wendy attempted to implement a principal components analysis (PCA) to reduce the dimensionality of the data. While this worked on a small test set, (9 3 by 3 images), when scaling to the full 10,000 we ran into the issue of overlong computation and running out of memory. Since the goal of PCA was to reduce runtime, this was undesirable. It was also difficult to test if the PCA was truly working, other than checking that the overall dimensionality of the data was reduced afterward.

Our KNN algorithm achieved an accuracy rate of around 35% (well above the random 10% threshold). However, given more, time we would attempt to achieve a higher success rate. There are several approaches given in the literature, including using a Weighted k-Nearest Neighbors (W-kNN) algorithm or a KNN algorithm modified with genetic algorithms. Both of these approaches would yield higher accuracy rates, but would require the investment of more time for us to appreciate the intricacies of some of the more complex CS involved.

The Kmeans algorithm worked well; however, Kmeans takes a significant amount of time to run. The reason being is that the dataset is quite large and that Kmeans must iterate over a significant portion of this data. Because of this long run time, it proved difficult to test its own functionality and later its accuracy because the algorithm would have to run for nearly 5 minutes for us to even identify an error or bug in the code, and then the randomness inherent in the code made it such that bugs did not crop up every time. Given more time, we would have optimized Kmeans using a different approach such as LLoyd's Algorithm or the streaming method in order to increase efficiency and to decrease run time.

Also given more time we would look into run-time optimizations. This would involve both adjustments such as possibly using C for some of the functions, as well as looking more into PCA to try to get that to work efficiently on the large dataset.

Advice for Future Students

We learned that it is very important to make sure all workspaces are entirely functional at the very beginning of the project. Though we had Python set up on all workspaces, we did not anticipate the issues that the appliance would cause by having multiple python paths. It is also important, if possible, that the workspaces are standardized so that there are no issues with compatibility among each other's code. Also, following the recommended timelines and implementing small pieces of the project piece by piece ahead of time gave us enough to time to implement the project successfully and indicated what parts of the project needed more work.