CS-512 Final Project
Supervised by :
Professor Antonio Miranda

Image Classification using k-Nearest Neighbours algorithm

GROUP 2:
ABHISHEK BHATT (ab2083)
HARSH BHATT (hb371)
SIVA HARSHINI DEV BONAM (sdb202)
TIANZHI CAO (tc796)

05-12-2020

Contents

1. Project Description

1.1 Introduction

In this project, our goal is to build an application for handwritten digit images classification using the *k-Nearest Neighbours (KNN)* algorithm[1]. KNN discovers "similar" items based on a pre-defined measure of "distance", for some pre-labelled data. Based on this measure for degree of similarity, the class to which a new, un-labelled data point belongs can be identified. The approach is also called *lazy learning* as there is no "actual" model training involved : computation with training data is deferred until the actual evaluation with testing data. Through this project, we implemented a known algorithm not covered in class, and explored its application to classify PNG images of handwritten digits. A simple web application was also designed, which serves as the user interface for the system.

1.2 Dataset

For this project, we used the MNIST database of handwritten digits[4]. It contains a training set of 60,000 examples, and a test set of 10,000 examples. Each image is 28x28 pixels.

1.2.1 Data Format

We used the training dataset provided by MNIST in the IDX file format as our "reference" to compute similarity with, for any given test image. For testing, we used the MNIST test dataset converted to PNG format (see Figure 1.1). The total size of training images and labels in IDX format is 47.1 MB when uncompressed. The total size of PNG test images is 2.5 MB when uncompressed. The data is at rest and can fit in a typical desktop.

1.3 Project Modules

This project has two high-level components. First part is the KNN algorithm implementation in Python. This module is the actual implementation of the KNN algorithm, which takes an image

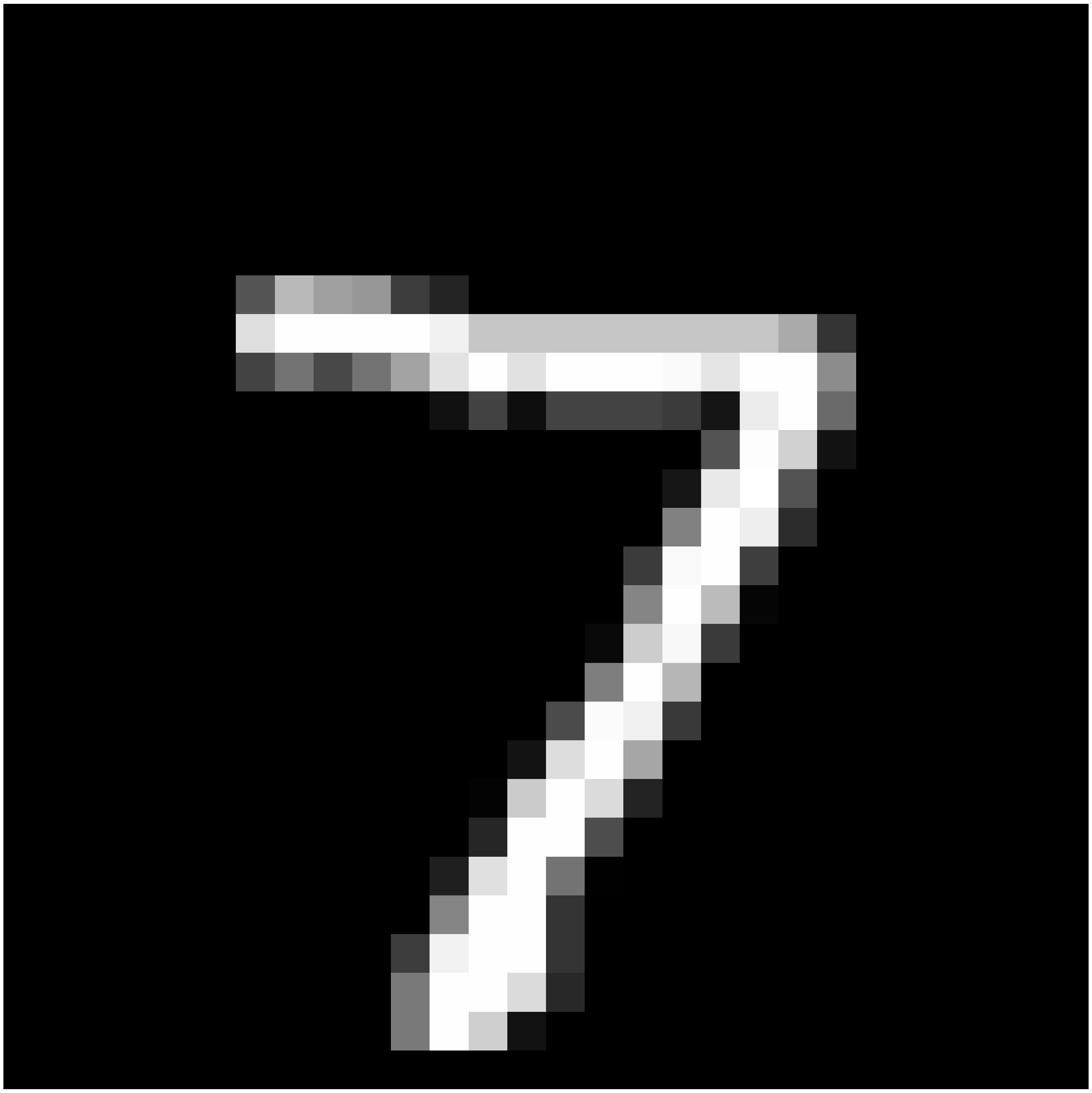


Figure 1.1: A test image in PNG format. Expected output from the implementation for this test image is 7.

as input and based on its "k Nearest Neighbous", predicts the handwritten digit in that image. The second component is a web application written using Python Flask. The application provides a user with a web-based (HTML) user interface. The interface allows uploading PNG images from the test set, and displays the corresponding digits each image contains. This module abstract away the kNN implementation and other application boilerplate code from the end user.

1.4 Project Goals

1. Implement KNN algorithm to find nearest neighbours (pre-labeled images in the MNIST training set) of an input image from the MNIST testing set.
2. Write an application on top of the KNN implementation that takes an image as input, and returns the digit it contains as an output.
3. Summarize the overall performance of the implementation in terms of time and accuracy.

2. Algorithm

KNN is a widely used algorithm in pattern recognition and data mining applications. Primarily, it is useful for answering two types of problems [3]:

- Classification : Given an object, to which class this object belongs. There are a finite number of pre-defined "classes" in this case for each data point.
- Regression : The output here is a single value for an input object. The value is computed as some aggregation of the values in the object's "neighbourhood".

2.1 Measure of similarity

In both the problems, the key subroutine is computing the k nearest neighbours for any given object. This means that for any given item, we discover the k items with minimum possible "distance" from the given item.

We use *Euclidean distance* as the distance measure for KNN. This formulation represents all data points as n-dimensional vectors in the Euclidean vector space (similar to Figure 2.1). In practice, we think of each data point as a collection of some values, with each value corresponding to some *feature* or property of that data point. The number of dimensions is equal to the number of such features. We use this idea of distance to ascertain how similar two data points are. The smaller the Euclidean distance between the two points, the higher is their degree of similarity.

For any two data points p and q, the Euclidean distance is then defined as:

$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2 + \cdots + (q_n-p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}$$

2.2 Distance formulation for images

In case of our implementation, each 28x28 gray-scale image is a data point (vector), with each dimension (out of 784 dimensions) representing a pixel in that image. The magnitude (value)

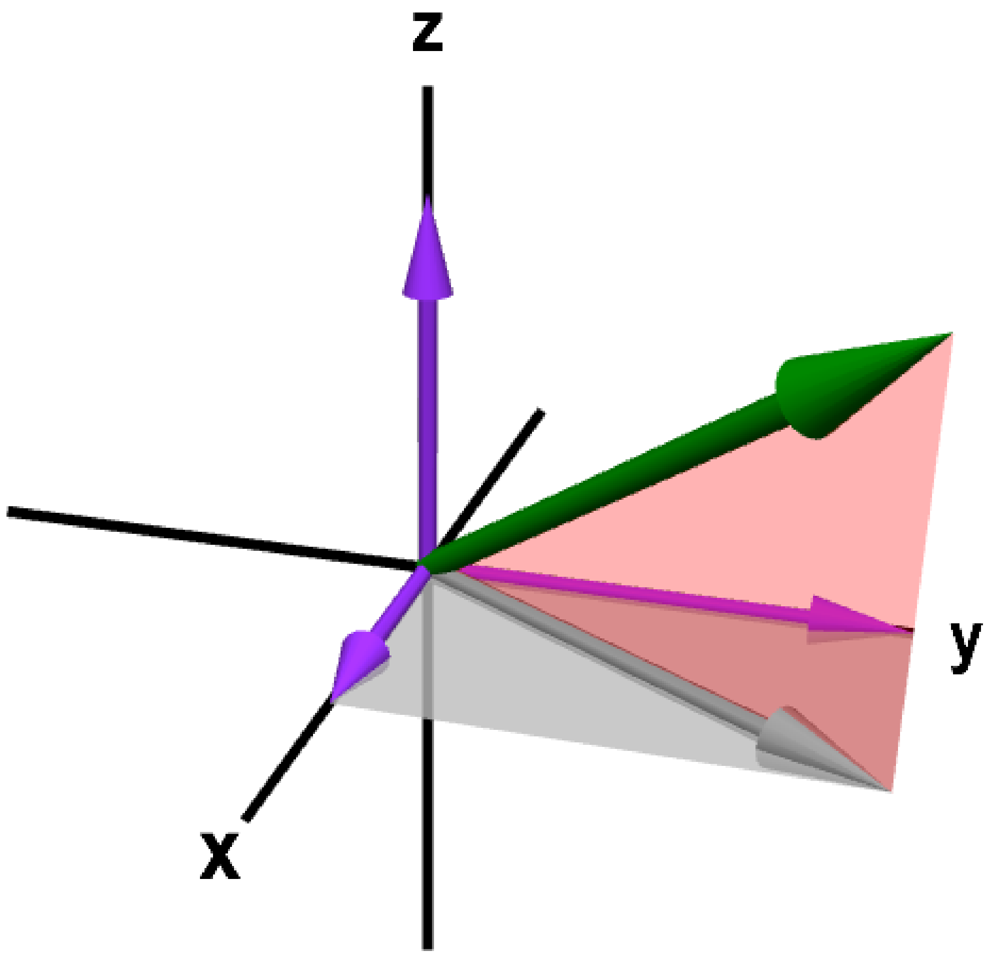


Figure 2.1: A 3-dimensional vector space.

along each dimension is thus the brightness of that pixel on a scale of 0 (black) to 255 (white), called pixel value.

Computing Euclidean distance between these image vectors therefore, gives a similarity measure between the corresponding images.

2.3 Nearest Neighbourhood

As described previously, we have a set of images with pre-defined labels, called the *training set*. There is another set of un-labelled images that form the *testing set*. Whenever, an input is given to the system from the testing set, the image is vectorized and its Euclidean distance is computed against the images in the training set. The number of train images to compute distance with is controlled by passing an additional parameter to the KNN classifier.

Once the Euclidean distances between the test image and the train images are known, we pick the k smallest of those values. The corresponding images in the training set, are thus the most similar to the given test image. These k train images collectively form the *k-Neighbourhood* of the test image, such that these k images are called the *k nearest neighbours* for the test image. Selecting the right value of k depends on the size of the dataset. A small value of k means that there are few reference data points to compute similarity with leading to low accuracy. A large value makes the KNN implementation computationally inefficient and defeats the motivation behind using KNN (points that are near might have similar densities or classes). For our implementation, we have used k = 3.

2.4 Image classification

Finally, we check the corresponding labels of the k nearest images and return the label with the maximum occurrence (see Figure 2.2). This return label is the *class* to which the input test image is assigned. In our implementation, the classes are digits from 0 to 9, which are returned as output for each input test image.

As a lazy learning algorithm, the model *learns* when the testing is performed. So there is not a distinct training step for the model. When one or more test images are uploaded from the UI, for each of the input images, labels for the k nearest neighbours of that test image are discovered from the train set, and the label (digit) with the most occurrence out of k is returned as the output for that test image.

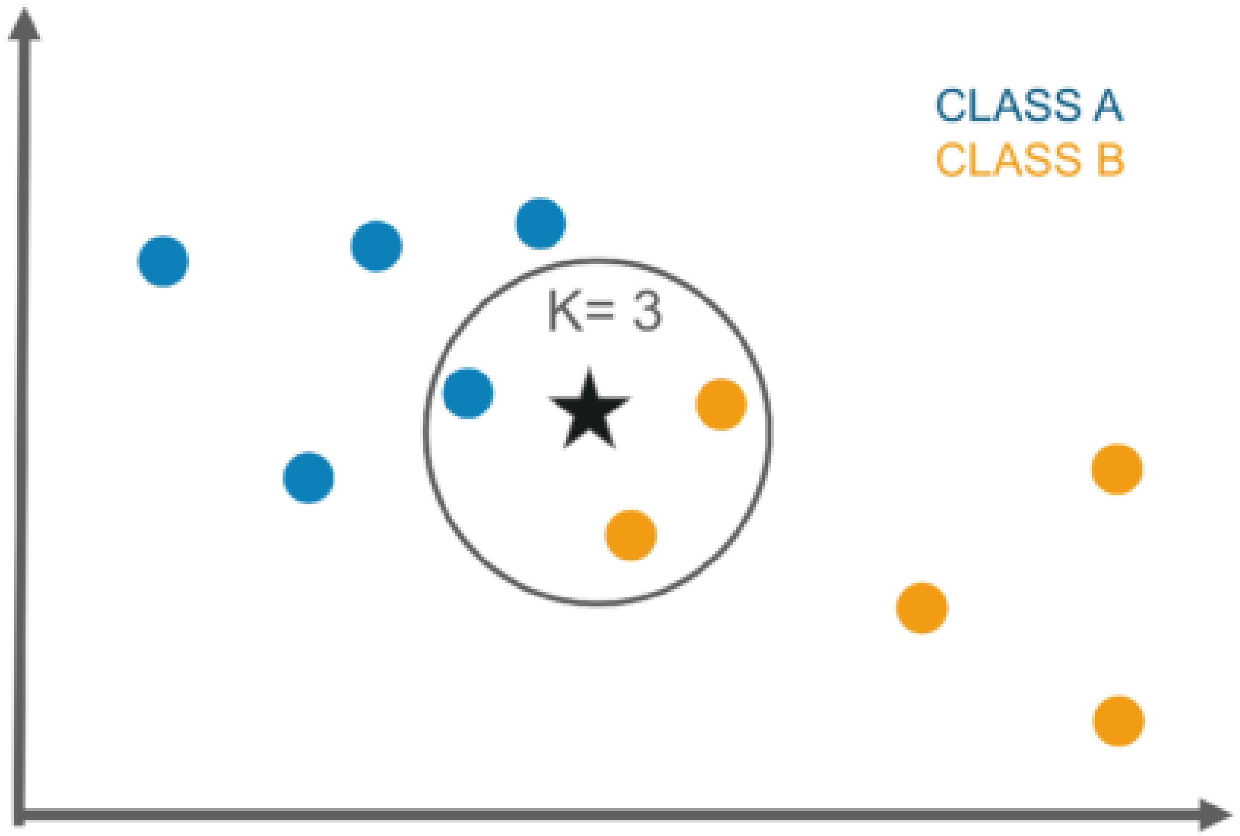


Figure 2.2: A 3-NN classifier.

3. Pre-processing for KNN

Although KNN is a relative simple algorithm, we still need to do some pre-processing in order to define some parameters for the model.

Firstly, to find the most appropriate K value, we tested for classification accuracy by setting values of k from 1 to 10. As clear from Figure 3.1, no significant change in accuracy was observed for k = 1 to 10. The running time for these different values is also roughly the same. For our implementation, we chose k=3. Taking a smaller value for k will decrease the likelihood of repetitive classes in the k-neighbourhood, and consequently we can prevent *over-fitting* of our model.

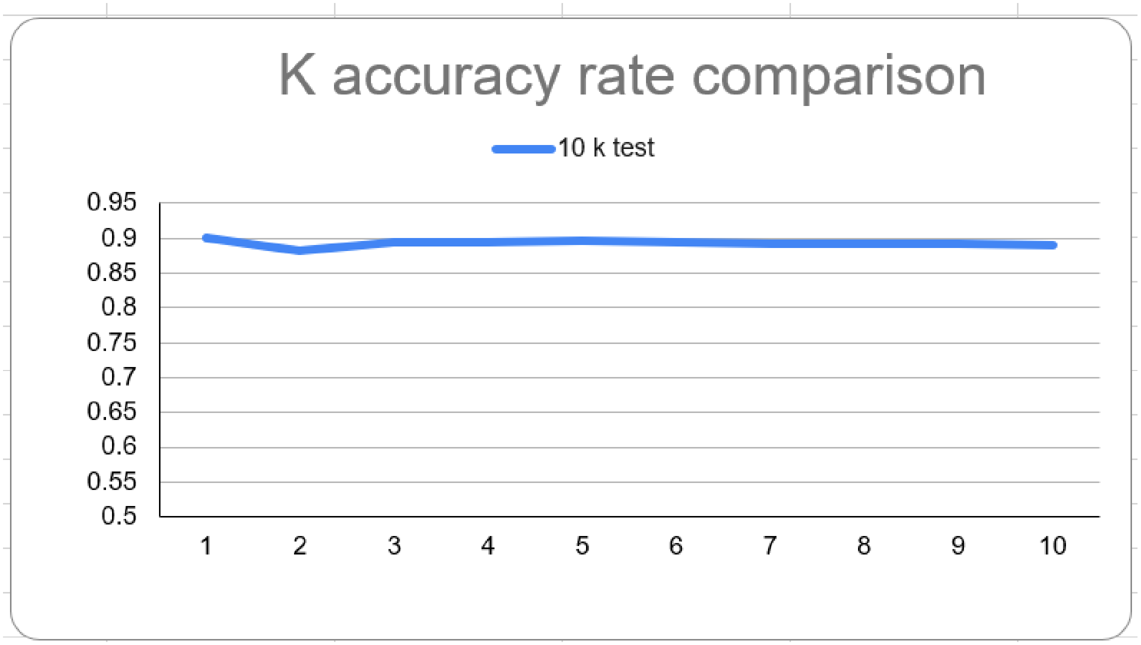
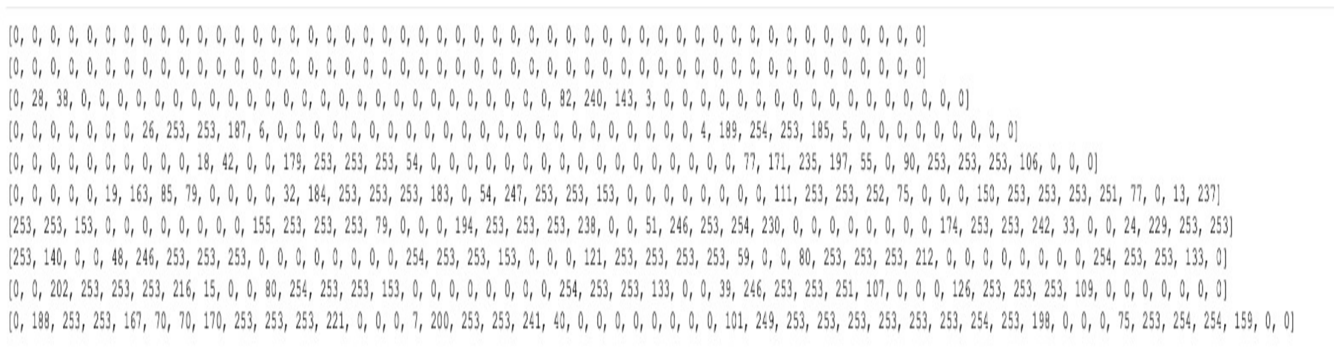


Figure 3.1

In the MNIST dataset, each image has 28x28 pixels which translates to 28x28 = 784 dimensions in the image vector (see Figure 3.2). Calculating Euclidean distance is very slow for 784 dimensional vectors. In order to improve the computational efficiency of our implementation, we sub-sampled both the training and testing images. The approach used for sub-sampling is removing one row of pixels after keeping a row, and removing one column of pixels after keeping a column. After sub-sampling, the 28x28 pixel images are reduced to 14x14 pixels each. Performing sub-sampling on the images is thus equivalent to dimensionality reduction of the image vectors.



```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 28, 38, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 82, 240, 143, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 26, 253, 253, 187, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 189, 254, 253, 185, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18, 42, 0, 0, 179, 253, 253, 253, 54, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 77, 171, 235, 197, 55, 0, 90, 253, 253, 253, 106, 0, 0, 0]
[0, 0, 0, 0, 0, 19, 163, 85, 79, 0, 0, 0, 0, 32, 184, 253, 253, 253, 183, 0, 54, 247, 253, 253, 153, 0, 0, 0, 0, 0, 0, 0, 0, 111, 253, 253, 252, 75, 0, 0, 0, 150, 253, 253, 253, 251, 77, 0, 13, 237]
[253, 253, 153, 0, 0, 0, 0, 0, 0, 0, 0, 155, 253, 253, 253, 79, 0, 0, 0, 194, 253, 253, 253, 238, 0, 0, 51, 246, 253, 254, 230, 0, 0, 0, 0, 0, 0, 0, 0, 174, 253, 253, 242, 33, 0, 0, 24, 229, 253, 253]
[253, 140, 0, 0, 48, 246, 253, 253, 253, 0, 0, 0, 0, 0, 0, 0, 0, 254, 253, 253, 153, 0, 0, 0, 121, 253, 253, 253, 253, 59, 0, 0, 80, 253, 253, 253, 212, 0, 0, 0, 0, 0, 0, 0, 0, 254, 253, 253, 133, 0]
[0, 0, 202, 253, 253, 253, 216, 15, 0, 0, 80, 254, 253, 253, 153, 0, 0, 0, 0, 0, 0, 0, 0, 254, 253, 253, 133, 0, 0, 39, 246, 253, 253, 251, 107, 0, 0, 0, 126, 253, 253, 253, 109, 0, 0, 0, 0, 0, 0, 0]
[0, 188, 253, 253, 167, 70, 70, 170, 253, 253, 253, 221, 0, 0, 0, 7, 200, 253, 253, 241, 40, 0, 0, 0, 0, 0, 0, 0, 0, 101, 249, 253, 253, 253, 253, 253, 253, 254, 253, 198, 0, 0, 0, 75, 253, 254, 254, 159, 0, 0]
```

Figure 3.2: Image vector representation.

KNN's time complexity is $O(d * n + k * n)$ [2], where n is the cardinality of the training set and d is the dimension of each sample. Therefore, it is expected that the running time should improve about 4 times after sub-sampling. However, by reducing the number of dimensions of each vector, we also introduce uncertainty in the data (due to some lost features), that might reduce accuracy of the model.

To get the optimum trade-off, we tested the performance of the solution both with and without sub-sampling, for 10%, 20% and 30% of the training set. The results are shown in figures 3.2 and 3.3 for this analysis, for running time and accuracy respectively. We observe that although the average correctness of predicted labels decreases about 2% after sub-sampling, there is a significant increase in the running times (more than 4 times as we include more data points from the training set). From this analysis, it is clearly justified to use sub-sampling in our implementation for improved running time, without significantly affecting the accuracy.

/KnnClassifierAnalysis/importpictest.py is for test performance of KNN. there are four parameters: -t, training set size (1-100) -d using sub-sampling or not (0 or 1), -k, K value (1-10), -s test set size (1-100). Testing the difference by using different training set can be done by run "bash run.sh"; testing the difference by using different k value can be done by run "bash runk.sh"; testing the difference by using sub-sampling or not can be done by run "bash runs.sh";

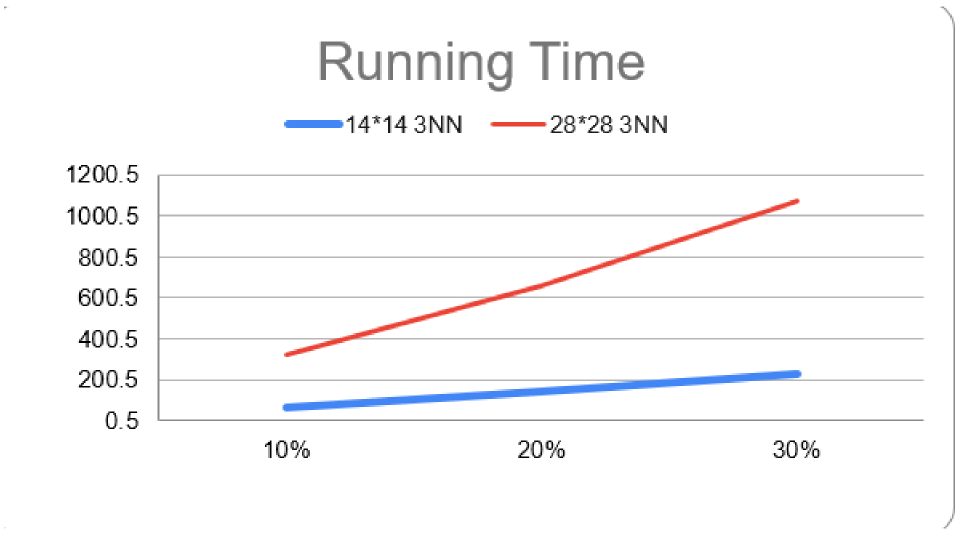


Figure 3.3

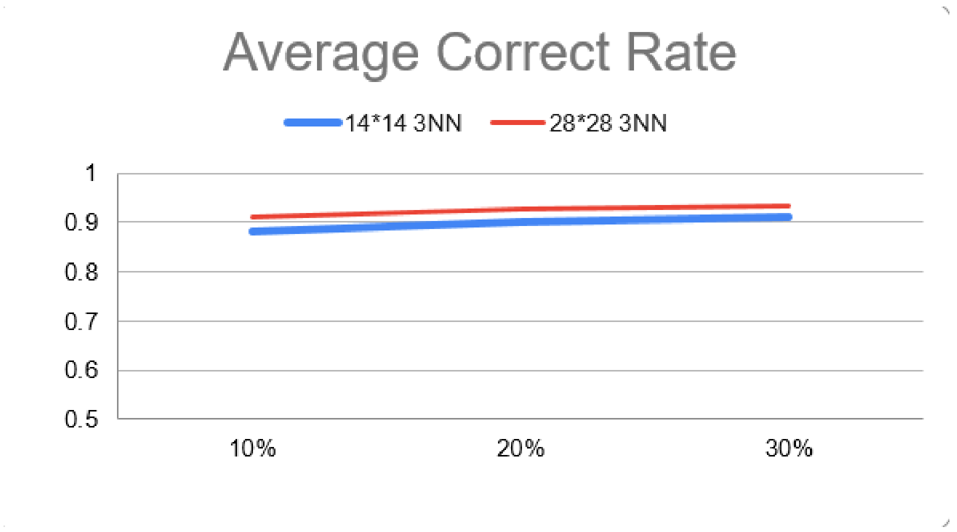


Figure 3.4

4. Implementation

Our solution is coded fully in the Python programming language (v3.5+). Some key Python libraries used for the implementation are:

- Pillow or Python Imaging Library (Fork) : To read each of the uploaded PNG files from the UI.
- Flask : To create a web application that can be accessed from a web browser.
- Numpy : For computations on matrix and/or vectors.

To deploy the web application, we need to run applicationknnimageclassifier.py with Python. The UI is then accessible at port 8080 on the localhost URL.

The code contains a Python Class named *KnnClassifier*, which contains the logic for determining the category (digit) to assign to an input test image, based on the k nearest neighbours of that image in the training set. The KnnClassifier instance takes the following parameters :

- trainingsize : Cardinality of the training set, or the number of images to pick out of the training set for which Euclidean distances are computed from each test image.
- traingingdata : Sub-sampled images read as a single 2D matrix.
- traininglabel : Vector of sub-sampled image labels.
- testdata : Single test image as a 2D array.
- k : Number of neighbours to consider for classification.

KnnClassifier is imported as a module in the *knn_image_classifier* web application. Input test images are uploaded from the UI and each of these images, say *itest*, is passed sequentially to an instance of the KnnClassifier. The classifier predicts the digit for each image and the corresponding result is collected in a dictionary. Once all uploaded images are processed, the collected results are returned to the UI. Figure 4.1 summarizes the process flow between the various components in the application.

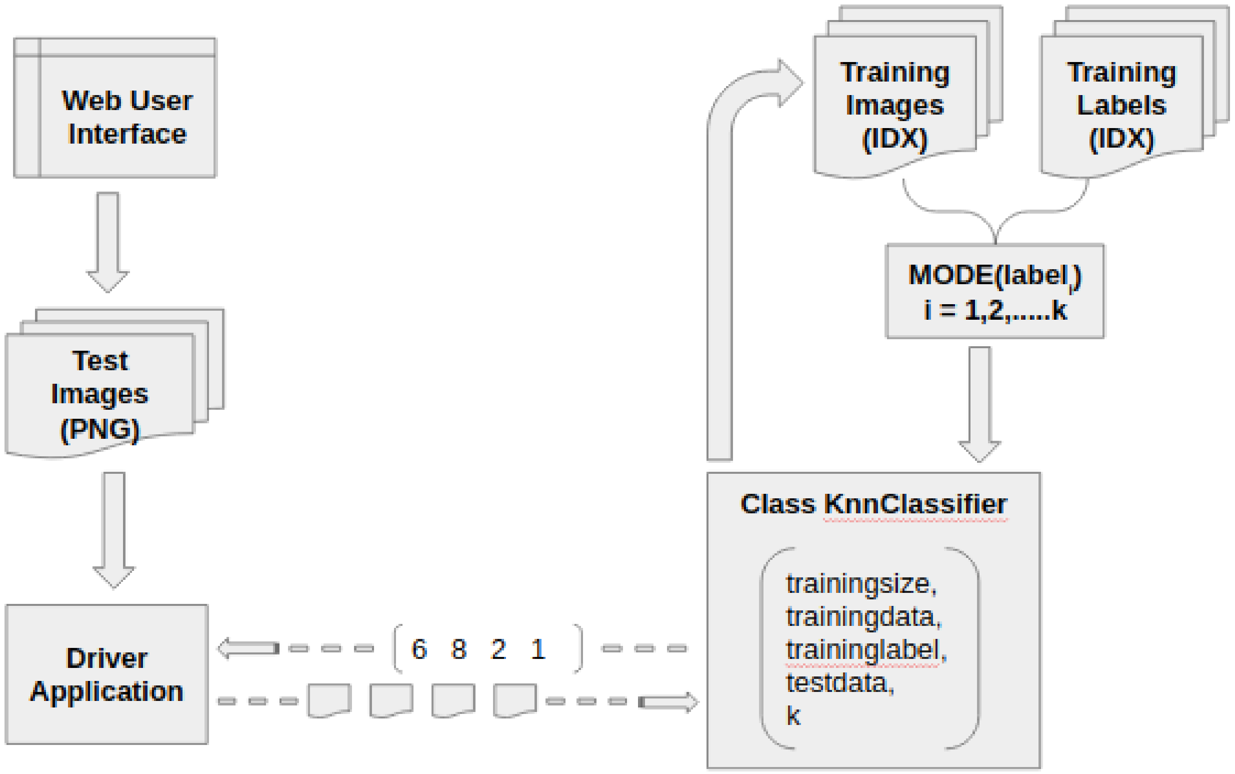


Figure 4.1: Process flow diagram.

5. Analysis and Conclusion

For analyzing the performance of our 3-NN classification algorithm we ran the application multiple times with 10 percent increments in the training set cardinality. The observations for the accuracy (also refer Figure 5.1) and time taken by the algorithm (also refer figures 5.2 and 5.3), for different sizes of the training dataset, are listed in the table below. Since all other operations in the application take approximately the same time overall, the plot obtained in figure 5.2 (total running time) has roughly the same shape as in figure 5.3 (running time for computing k nearest neighbours).

k=3	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
KNN Accuracy	0.8804	0.9018	0.9108	0.9144	0.9206	0.9232	0.9282	0.914	0.9344	0.9374
Total Time	63.5	146	228.5	309.3	403.3	495.2	596.9	772	932.7	1064.8
Calculation Time	1.785	4.101	6.47	9.005	11.683	14.724	18.57	23.973	28.357	32.746
Total Time Increase	63.5	82.5	82.5	80.8	94	91.9	101.7	175.1	160.7	132.1
Calculation Time Increase	1.785	2.316	2.369	2.535	2.678	3.041	3.846	5.403	4.384	4.389

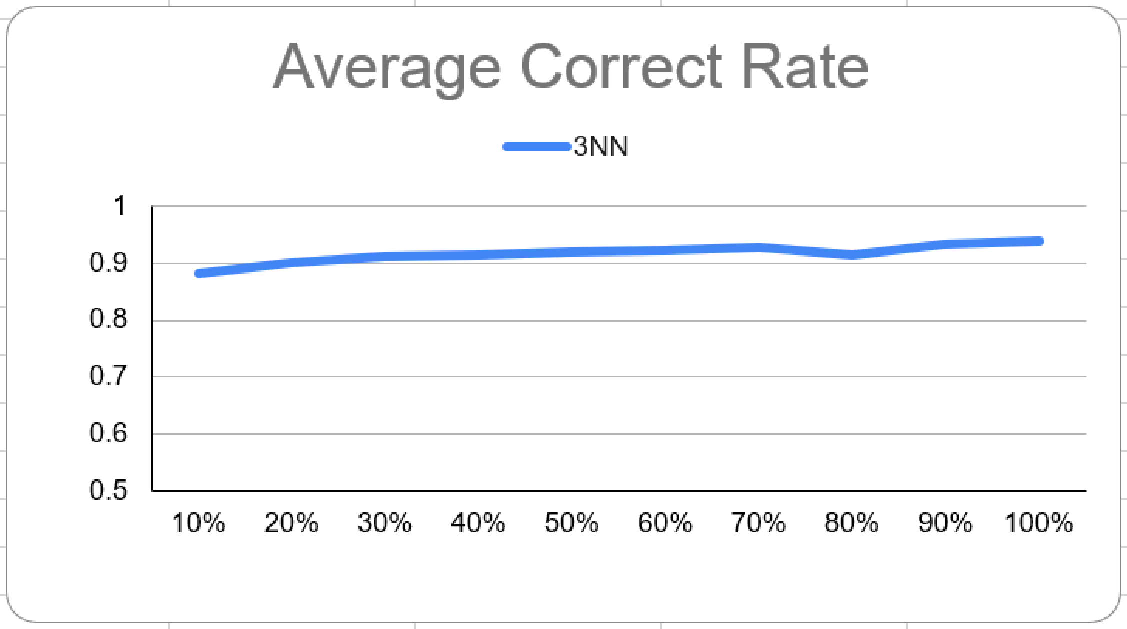


Figure 5.1: Correct Rate for k=3.

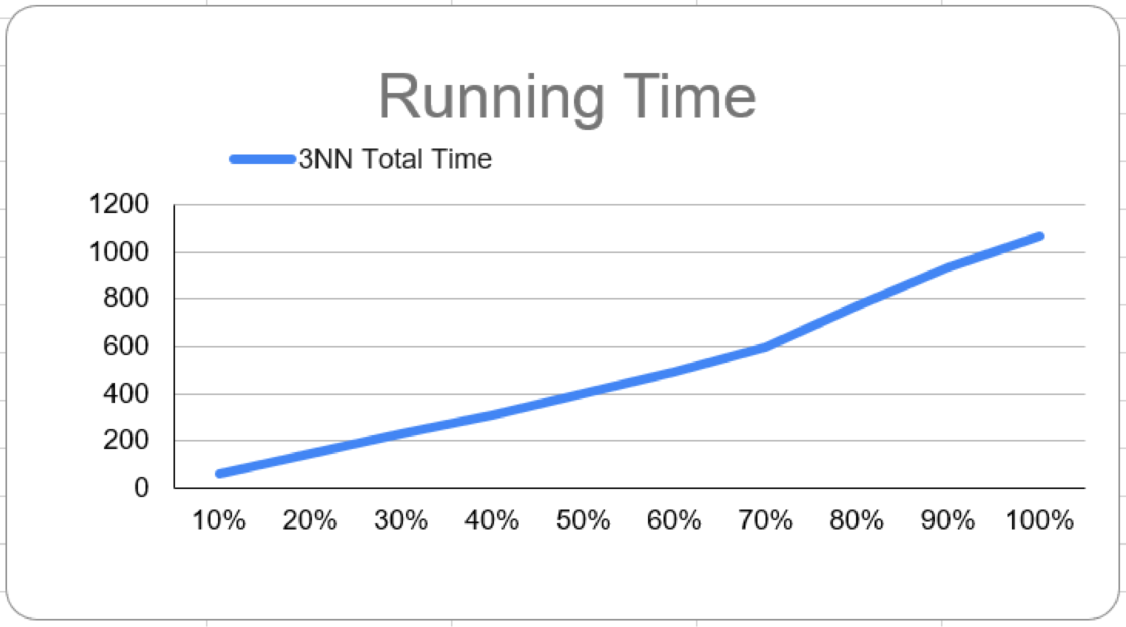


Figure 5.2: Total Running time.

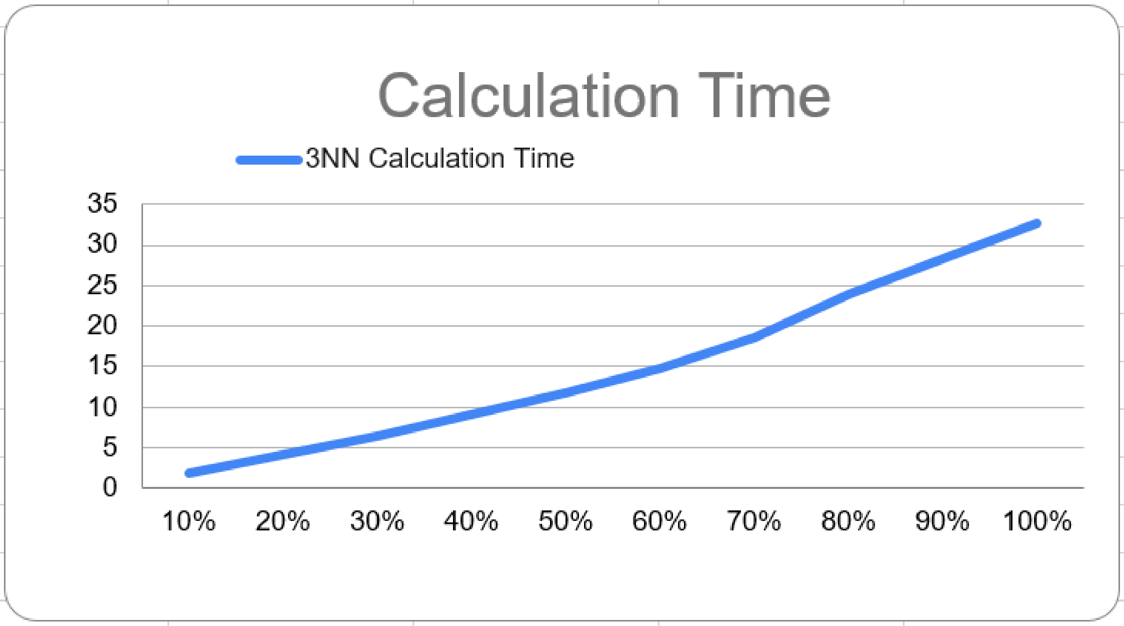


Figure 5.3: Calculation time.

With no optimization in KNN implementation (for example, using the K-dimensional tree data structure), the time complexity for KNN should be $O(d*n+k*n)$, where n is the cardinality of the training set, d is the dimension of each sample, and k defines the size of the neighbourhood for a given test data point. We conclude from our analysis, that as we increase the size of the training set by 10% increments, the running time also increases approximately linearly until we take 70% of the training set. After that the slope rises indicating larger change per increment. Our understanding is that beyond 70% of training size, the computing resources are overwhelmed by the k nearest neighbours computation and thus the computation awaits RAM getting freed. As a result, the overall running time increases sharply.

Thus, through this project we have successfully demonstrated the application of the K-Nearest Neighbours algorithm for image classification and analyzed its performance in terms of the size of the training dataset.

Bibliography

[1] Adi Bronshtein. A quick introduction to k-nearest neighbors algorithm, 2017.

[2] Stack Overflow community. k-nn computational complexity, 2020.

[3] Wiki community. k-nearest neighbors algorithm, 2020.

[4] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits, 2020.