CS420 Project Report

Ruiheng Chang
515021910459
Department of Computer Science
Shanghai Jiao Tong University
crh19970307@sjtu.edu.cn

Weichao Mao
515021910559
Department of Computer Science
Shanghai Jiao Tong University
maoweichao@sjtu.edu.cn

1 Traditional Models

In this section, we first preprocess the images on the pixel level to remove the disturbances. We then perform traditional classification models, such as KNN and SVM, on the processed images. We further compare the performance of the traditional models with and without the preprocessing procedure. Finally, we employ ensemble methods and try to obtain better predictive performance.

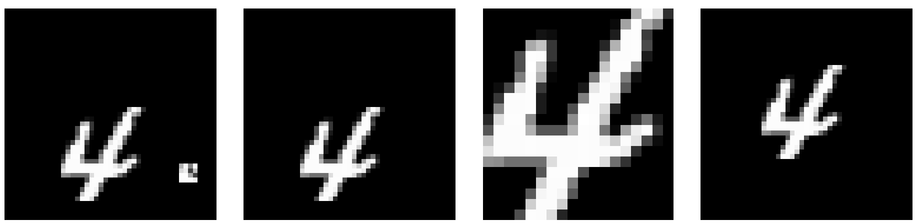
1.1 Pixel-Level Image Preprocessing

We can easily see the data set given to us is generated by adding some minor disturbances to the standard MNIST data set. These disturbances include shifting of the main digit, and small random pepper noise which presents itself as sparsely occurring white pixels.

Before applying traditional machine learning algorithms on the disturbed MNIST data set, we first perform pixel-level preprocessing on the images, aiming to reverse the disturbances and reconstruct the original images. In the following, we will demonstrate our preprocessing procedure step by step.

1.1.1 Timeline Overview

The preprocessing steps are summarized in Figure 1. Figure1(a) shows one original image from the given data set. As we can see, there is a patch of white noise on the right side, and the digit is located in the lower half of the image. Our first step in the preprocessing removes the white noise on the right, and the denoised result is shown in Figure1(b). In the second step, we crop out the minimum rectangle that covers the digit pixels, and Figure 1(c) shows the cropped digit. Finally, Figure 1(d) shows the centered image after we add equal padding to the digit.



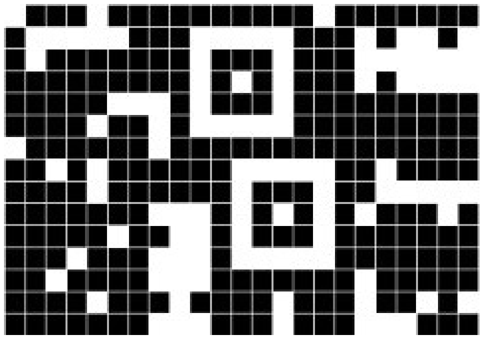
(a) Original (b) Denoised (c) Cropped (d) Centered

Figure 1: Preprocessing timeline overview.

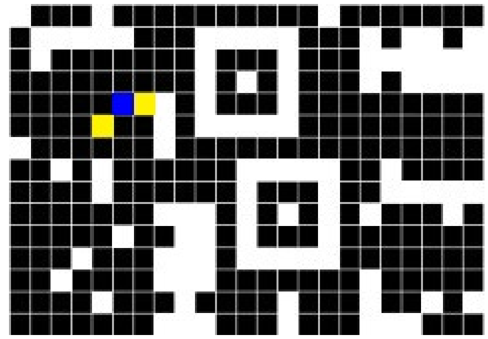
1.1.2 Step 1: Denoising

The first step concerns searching and removing the white pepper noise in the image. In this step, we regard any white pixel block with no more than 20 pixels as the noise area. The intuition is, the pixels of digits 0 to 9 are all connected blocks, and since the digit is the main part in the image, it must contain a large area of pixels (more than 20 pixels).

We utilize the simple Depth First Search (DFS) algorithm to search for the connected blocks. We enumerate each of the pixels in the image from the upper-left corner. If the current pixel is a white pixel, we will further check its 8 neighboring pixels and if there exist white pixels among its neighbors, we connect them into a single block. Figure2 demonstrates the detailed process of this step. Suppose the original binary image is shown in Figure2(a). For a white pixel we are visiting (colored in blue in Figure2(b)), we need to further check its 8 neighbors. Since we find two more white pixels among its neighbors (colored in yellow in Figure2(b)), we connect the three pixels into a single block, and repeat the same procedure on the two neighboring white pixels. Finally, we will find the whole white connected block.



(a) Original image



(b) Neighbors

Figure 2: White block searching.

Once we have found the noise area (i.e., the white connected blocks with area no more than 20 pixels), the denoising process is easy to implement. We only need to employ the Flood Fill algorithm on the noise areas, and color the white pixels into black. A different statement of the same process is to perform a second DFS on the noise pixels, and color every white pixel we visit into black.

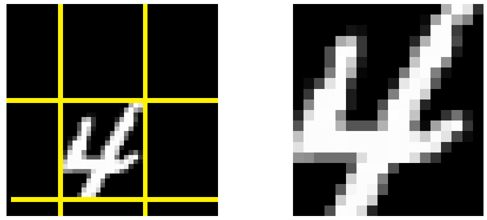
1.1.3 Step 2: Cropping

So far, we have safely removed the white pepper noises from the images, but we still cannot feed the current digit into the traditional models directly. The problem is, the digits are not centered in the image, and although some models like convolutional neural networks are not sensitive to shifting, these misaligned digits can rule out many traditional models like KNN. To guarantee a reasonable performance of the traditional models, we further need to center the digital pixels right in the middle of the image.

This second step is relatively easy. We only need to find the boundaries of the digit and crop it out. As illustrated in Figure3, we can enumerate to find the upper-most, left-most, right-most and lower-most pixels in the denoised image, and use these pixels as boundaries (denoted by yellow lines in Figure3(a)) to crop out the digital area.

1.1.4 Step 3: Centering

Finally, we will center the digital area right in the middle of the image. We want to keep the size of the image (45×45) unchanged before and after the preprocessing procedure so that we can simply feed the new



(a) Boundaries (b) Cropped digit

Figure 3: Cropping out the digit.

data set into our previous models. Since the size of the cropped image can be a little smaller than the original image, we add equal padding to the left, right, up and down side of the digit to make its size equal to the original one. The equal padding also guarantees the digit will always be centered in the image, so that the misalignment of digits is not a problem anymore. The OpenCV library provides a function *copyMakeBorder* that implements this operation. Therefore, we only need to calculate the padding length and then safely rely on OpenCV to do the padding job.

So far, we have finished our preprocessing steps, and the images are ready to be feed into the traditional models. For more detailed explanations of these preprocessing steps, please refer to the comments in our code in file `preprocessing.py`.

1.2 K-Nearest Neighbors

In this part, we utilize the simplest classifier, the k-nearest neighbors algorithm, or KNN, on the given data set as well as our processed data set. KNN is a non-parametric model. An test data sample is classified by a majority vote of its neighbors. More specifically, the data point is assigned to the class most common among its k nearest neighbors.

The reason we choose this algorithm is that we are clear KNN is very sensible to the shifting of the digits in the image. By comparing the KNN output on the given data set with its output on our processed data set, we can easily check whether our preprocessing step is helpful or not, and to what extent the preprocessing helps improve the performance.

Before we perform KNN on the data sets, we first refer to Principal Component Analysis, or PCA, to reduce the dimensionality of the data sets. We will not dig into the theoretic details of PCA (and KNN) in this report since they are well covered in the course lectures. All we need to care about is the specific PCA ratio will have a significant influence on the performance of our models. The scikit-learn library provides the PCA and KNeighborsClassifier classes so that we can safely rely on these modules to achieve our goal. For more detailed implementation of this part, please refer to our codes in file `KNN_without_preprocessing.py` and `KNN.py`.

Throughout our experiments, we use the simple hold-out validation, and randomly select 30% of the training samples as our validation set. We test the KNN performance of different setting of parameters on the validation set, and select the parameter values with highest accuracy to feed into the model on the test set.

1.2.1 Influence of PCA Ratio

In this part, we test the influence of PCA ratio on KNN. We enumerate the PCA ratio value from 0.4 to 1.0, and for each PCA ratio value, we enumerate the value of k on the validation set to find the optimal value under the current PCA ratio. Since this nested iteration process is relatively time consuming, we only use 10000 training samples and 3000 testing samples in this part. From Homework 3 we know that the size of training samples will indeed influence the performance of our model, but we believe the trend of the performance will not vary much. The performance of KNN on the given data set and our processed data set under different PCA ratio values is shown in Figure 4. The values of k for KNN are separately optimized for the two validation set under each specific value of PCA ratio.

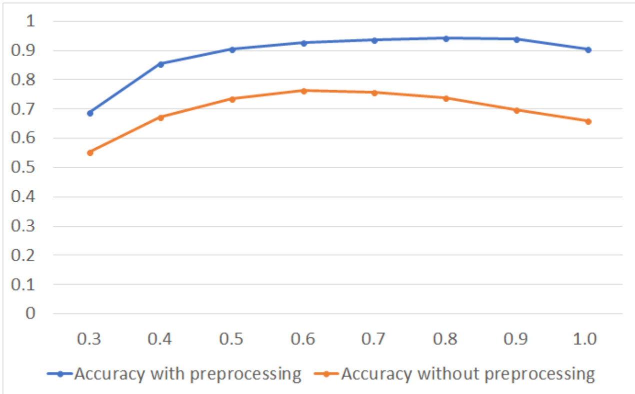


Figure 4: KNN performance under different PCA ratios.

As we can see, for KNN both with and without preprocessing, the accuracy first increases with PCA ratio and then decreases. The highest accuracy of KNN with preprocessing is achieved at PCA ratio equals 0.8, and for KNN without preprocessing the optimal PCA ratio is 0.6.

1.2.2 Performance under Optimal Parameters

In this part, we feed all the training samples into the model, and use all the test data in the testing phase. For KNN with preprocessing, the optimal parameter settings PCA ratio and $k =$ lead to the highest accuracy of . For KNN without preprocessing, the highest accuracy is 8888 achieved at PCA raio = and $k =$. As we can see, since KNN is sensitive to spatial shifting, our preprocessing steps significantly improve the performance of KNN.

References