Nearest neighbors and handwritten digit classification

Andrea Favia(1498010)
a.favia@student.tue.nl

Mayank Padhi (1432788))
m.padhi@student.tue.nl

Núria Casals Lladó(1513656)
n.casals.llado@student.tue.nl

Paolo Berizzi(1518798)
p.berizzi@student.tue.nl

Eindhoven University of Technology — March 9, 2020

1 Introduction

Problem

This report aims to address the problem of handwritten digit classification. Our dataset contains a low resolution images of a handwritten digits. Then, our goal is to classify a new image as one of the digits in $\{0, 1, ..., 9\}$. More specifically, each image has 28×28 pixels and each pixel takes values in $\{0, 1, 2, ..., 255\}$. Therefore $\mathcal{X} = \{0, 1, ..., 255\}^{2828}$ and $\mathcal{Y} = \{0, 1, ..., 9\}$.

The dataset we are using is a classical dataset frequently used to demonstrate machine learning methods, known as MNIST dataset. The training set consists of 60000 images of handwritten digits and the corresponding label. The test set MNIST of 10000 images of handwritten digits and the corresponding labels.

The data has the setting of supervised learning. Each row represents one image and has 785 entries. The first entry in the row is the label, and the $784 = 28 \times 28$ subsequent entries encode the image of the digit each entry corresponding to pixel intensity.

The ultimate goal of this project is to build a classifier based on the k-NN rule which minimizes the probability of making an error. Even though the classification rule seems simple, we aim to find the best parameters to predict our current test data, as well as having a smart implementation to ensure this can be done quickly.

Methodology

To get a first-hand experience with the k-NN method, we first explored a subset of the original MNIST dataset. This consist of a train part of 3000 examples and a test part of 1000 examples. Doing so allowed us to implement the model and choose a good set of tunable parameters.

Hence, we first experimented with different k values. The second we did was to run the model with different k as well as different distances from the class of Minkowski distances. Last, we tried to broaden our perspective trying out other distances and also other preprocessing methods.

After the first iteration, we used the best parameters obtained from the small dataset to apply the algorithm on the entire dataset. At this point in the implementation, we had to cope with some optimization issues, since our program was not able to deal with the whole data.

Finally, we conducted a Principal Component Analysis of our data. This study helped us to reduce the dimensionality of the data without losing a lot of information. We then conducted the same analysis as before with this transformed data.

2 k-NN implementation with small MNIST dataset

The performance of the k-NN method relies on the choice of two parameters:

- k: the number of neighbors used for prediction
- $d : \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R}$, the distance metric used to define proximity of two points in the feature space.

Given a k and a distance d, the k-NN rule is defined as follows:

$$\hat{f}(x) = argmax_{y \in Y} \Big\{ \sum_{i=1}^{k} \mathbf{1}\{Y_{(i)}(x) = y\} \Big\} \tag{1}$$

Where $Y_{(1)}(x), \ldots, Y_{(k)}(x)$ are the classes of the k-nearest neighbors for a given x, measured with the distance d.

Moreover, we use the 0/1 loss function for the whole study, defined as follows:

$$l(\hat{y}, y) = \begin{cases} 1 & if y \neq \hat{y} \\ 0 & otherwise \end{cases}$$

So, if the predicted and true labels are identical we have zero loss, otherwise we have loss taking the value 1.

2.1 Euclidian distance

There are many possible choices for the distance metric used by kNN d, and a naive starting point is to consider the usual Euclidean distance. Given two points of $a, b \in \mathbb{R}^m$, the Euclidian distance is defined as $d(a, b) = \sqrt{\sum_{i=1}^{m} (b_i - a_i)^2}$.

How do we break ties?

According to equation 1, there might be situations where two (or more) classes appear an equal number of times. Ties occur when the same number of neighbors from 2 or more classes are around the data point.

There are several possible solutions to break ties. First, we can lower the value of k if a tie occurs. Since ties cannot happen for 1 neighbor, this solution always works. However, for larger datasets, this solution might be computationally expensive.

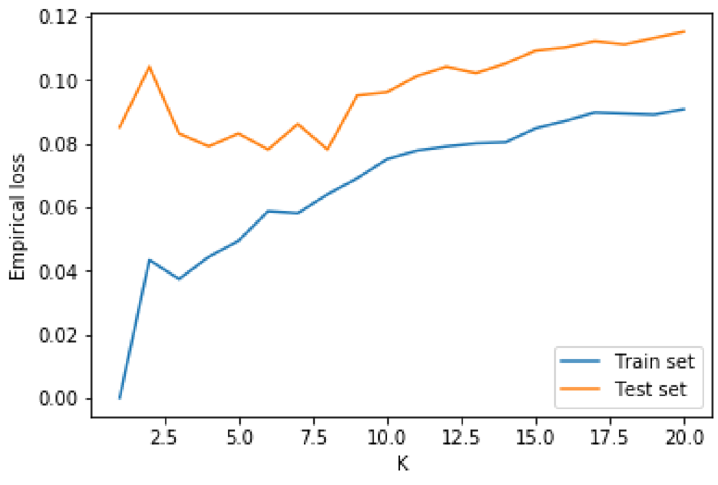
Second, randomly selecting a class out of the contending classes can also be a solution. However, this does not guarantee an optimal solution as well.

Another possible approach involved adding weight to the distances. The only pitfall of this method is that it may not work if all the points are equidistant from the given vector. Our approach is to use the first occurrence of the class on the list. That is, we consider the class that was first encountered while determining the class.

Results

We computed the empirical loss by averaging the loss value of all the examples, $\sum_{i=1}^{n} L(\hat{y}_i, y_i)$, where n is the size of either the train or test dataset. Hence, since we are using the 0/1 loss function, the resulting empirical loss is the ratio of misclassified examples.

In figure 1, we can observe that for $k = 1$, the train dataset has an empirical loss of 0.000. This is because given a point x, the nearest neighbor is x (trivial since $d(x, x) = 0$ for all the distances d). Hence, for $k = 1$ we are always going to predict the class correctly. Since the rule classifies according to the nearest neighbors of the training dataset, this does not happen for the test dataset. We can use the same logic to explain why for small k, the empirical loss differs significantly between test and train set. This observation motivates the next section, where we used the Leave-One-Out cross-validation.



k	Train loss	Test loss
1	0.000	0.085
2	0.043	0.104
3	0.037	0.083
4	0.044	0.079
5	0.049	0.083
6	0.059	0.078
7	0.058	0.086
8	0.064	0.078
9	0.069	0.095
10	0.075	0.096
11	0.078	0.101
12	0.079	0.104
13	0.080	0.102
14	0.080	0.105
15	0.085	0.109
16	0.087	0.110
17	0.090	0.112
18	0.089	0.111
19	0.089	0.113
20	0.091	0.115

Figure 1: Empirical loss for train and test set using Euclidean Distance for $k \in \{1, \ldots, 20\}$

Moreover, we see that for big values of k, both train and test loss tend to increase. This fact is a bit intuitive to predict because the larger k, the larger the number of neighbors the method uses to make the prediction and it leads to increase in the error.

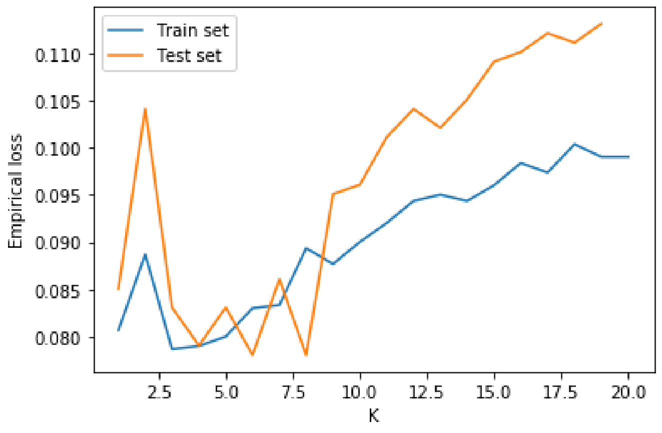
2.1.1 Leave-One-Out Cross-Validation

To get a better risk estimation, we implemented the Leave-One-Out Cross-Validation (LOOCV) method. In our setting, doing so is quite simple. To predict the class of the point x_i, we just remove x_i from the list of $k-$nearest neighbors. Doing so allows us to generalize better the train risk estimation because the method is no longer making use of privileged information like the class of the point we want to classify.

In figure 2, we can see the results of our implementation. We used LOOCV method for the train dataset, and plotted it with the empirical loss on the test dataset that we showed in the previous section.

Given these results, a good choice of k would be between 3 and 8. In this range of values, both LOOCV risk estimation and the test empirical loss are low.

We observe that the LOOCV risk estimation is significantly higher than the empirical training loss computed in the previous section This fact confirms that in the previous section we had an advantage when predicting



kLOOCV	Risk estimate
1	0.081
2	0.089
3	0.079
4	0.079
5	0.080
6	0.083
7	0.083
8	0.089
9	0.088
10	0.090
11	0.092
12	0.094
13	0.095
14	0.094
15	0.096
16	0.098
17	0.097
18	0.100
19	0.099
20	0.099

Figure 2: LOOCV risk estimates and test empirical loss using Euclidean Distance for $k \in \{1, \ldots, 20\}$

the classes in the train set. Therefore, loss estimation has increased. Moreover, and, the LOOCV risk estimation is significantly more similar to the empirical test loss, so indeed we get a more accurate result that in the previous section.

2.2 Minkowski distances

The distance metric plays a fundamental role in the knn-algorithm because it is the function that defines similarity and dissimilarity between different points and, in our case, between different images. The naive choice is the common and previously illustrated Euclidean distance, but we also investigated the generalization of this distance by considering the class of Minkowski distances. Given two points $a, b \in \mathbb{R}^l$ and value $p \geq 1$, the Minkowski distance is defined as:

$$d_p(a,b) = \left\{ \sum_{i=1}^{l} |b_i - a_i|^p \right\}^{\frac{1}{p}} \tag{2}$$

Based on the choice of $p \in \{1, \ldots, 15\}$, we defined a set of possible Minkowski distances and we tested each one of these with *leave-one-out cross validation* for all the possible value $k \in \{1, ..., 20\}$. As a result, we obtained the scores for all the combinations of p and k, which are presented in Figure 6.

We used the heatmap presented in Figure 6 to search for the best combination of p and k. The heatmap clearly shows the influence on the performance of the knn-algorithm given by both parameters p and k. Considering the effect of the choice of p, which means analyzing the heatmap vertically from the top to the bottom, it is possible to see a general improvement of the classifier as p increases defined by the reduction of the average loss of the *leave-one-out cross validation*. It is possible to explain this trend if we consider the high number of dimensions that characterize the dataset. When comparing data points with a high number of features, in our case $28 \times 28 = 784$, using simple distance metrics (eg. Manhattan distance, Euclidean distance) can be inappropriate compared to and increased value of p for the Minkoswki distance.

In the same way, analyzing the heatmap from left to right we can deep in the effect of the number of neighbors that are considered to classify a new image, the k parameter. At first, it is possible to notice

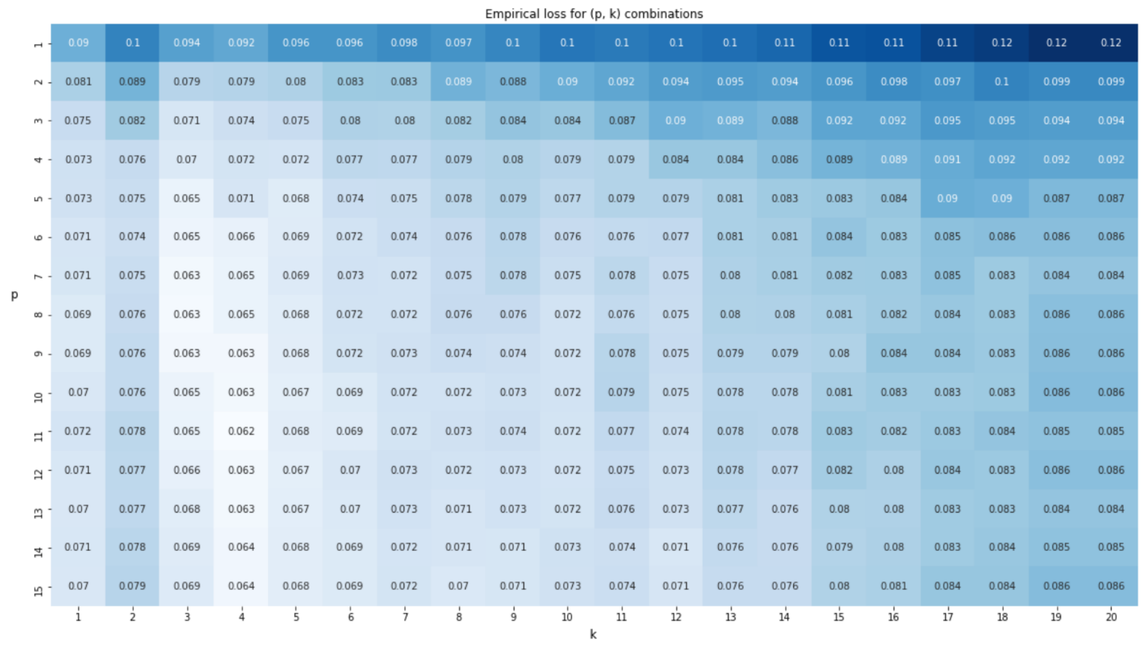


Figure 3: Loss heatmap for various $p \in \{1, \ldots, 15\}$ and $k \in \{1, \ldots, 20\}$

a clear step between the first and the second column, where the second one seems to perform worse. Proceeding in the increase of the number of neighbors, for $k \in \{3, 4, 5\}$ there is a slot of optimal values with the best performances obtained for $k = 3$ or $k = 4$. With the further increase of the number of neighbors considered by the classifiers the performance start to slowly decrease. This is explained by the fact that having many neighbors that is too high will end up in considering more neighbors of other classes and this will affect negatively the performance of the classifier.

In the end, the best classifier we identified is given by using a Minkowski distance with $p = 11$ as a distance metric and considering the first $k = 4$ nearest neighbors to predict the class. In that case the average loss obtained during the leave-one-out-cross-validation is 0.062.

2.3 Metric and preprocessing experimentation

Metric

The distance between the pixel values is calculated and stored in a matrix. Several distance metrics are used and the result obtained by conducting *leave-one-out cross validation* are later compared to discover the most appropriate one to get the best performance of the kNN algorithm. We further discuss the distance metrics that were used apart from the general Minkowski distance metrics previously discussed.

Minkowski distance, when used with p being 1 or 2, corresponds to the Manhattan distance and the Euclidean distance respectively. In the limiting case of p reaching infinity, we obtain the Chebyshev distance.

Manhattan Distance: Also known as the l_1 norm, the distance between two points is the sum of the absolute differences of their Cartesian coordinates. It can also be interpreted as the sum of the lengths of the projections of the line segment between the points onto the coordinate axes. Analytically, it is defined as

$$\sum_{i=1}^{n} |b_i - a_i| \tag{3}$$

We see that as we increase the value of k from 2, the loss decreases a bit to 0.094 and then increases again to 0.1 and remains almost constant till $k = 20$.

Chebyshev Distance: Also known as the l_∞ norm, it is defined in the way that the distance between two vectors is the greatest of their differences along any coordinate dimension. It examines the absolute magnitude of the differences between the coordinates of a pair of objects. Mathematically, it can be expressed as

$$max(|b_i - a_i|) \tag{4}$$

The loss is overall the highest among all the distance metrics. The loss first decreases from 0.45 (for $k = 2$) to 0.41 and then remains with increase in k with small changes at $k = 11$ and $k = 19$.

The following graph shows the performance of kNN with different distance metrices viz. Euclidean Distance, Manhattan Distance, Chebyshev Distance and Minkowski Distance (with $p = 11$) for $d = 2$ to $d = 20$.

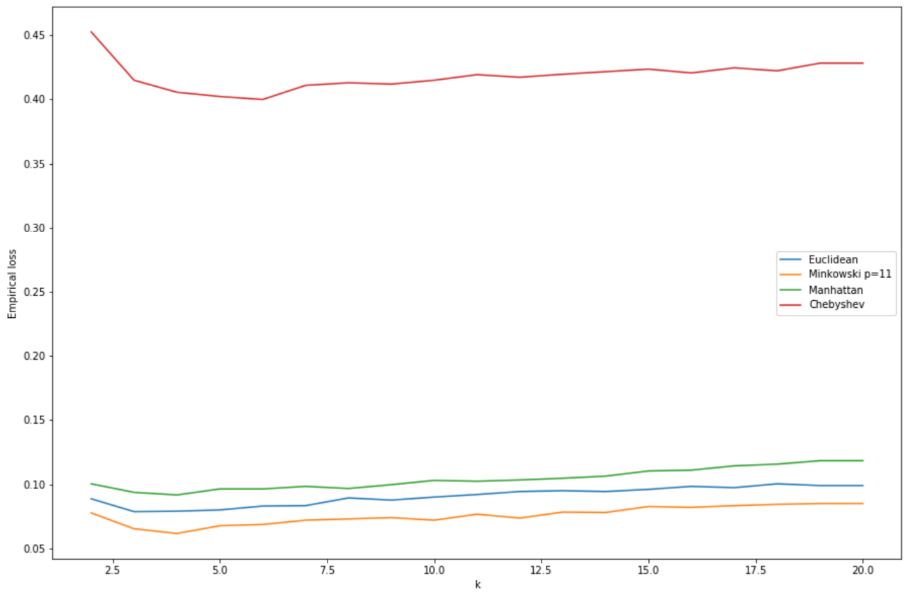


Figure 4: Performance of k-NN with different distance metrices

It is can be seen that Minkowski distance with $p = 11$ has the best performance for kNN. The Chebyshev distance has the highest loss. It can also be observed that the loss decreases at first and then has a slightly increasing trend after $k = 5$ for all the metrics.

Preprocessing

Preprocessing in images is important since it can help machine learning algorithms to learn better and faster. Images can present noise, have different sizes and pixel values must be treated somehow.

For our case, as a first step, we decide to scale in the images inputs in the range [0,1]. When using KNN it is recommended to have features in the same range otherwise some features will have more importance compared to others. However, it is worth notice that our features are already in the same range [0,255] and, as expected, we do not see any difference. We tested the scaling technique on the training data by using *Leave One Out* cross-validation for different values of k neighbors.

Later, we tried some techniques to smooth the images: Convolution and Image Blurring. Both of them are low-pass filters and hence useful in removing noise or blurring the image.

In this convolution case, we used an averaging 5×5 kernel: a 5×5 window is centered for each pixel and, after the summation of all the pixels, we compute the average for that precise window. The kernel

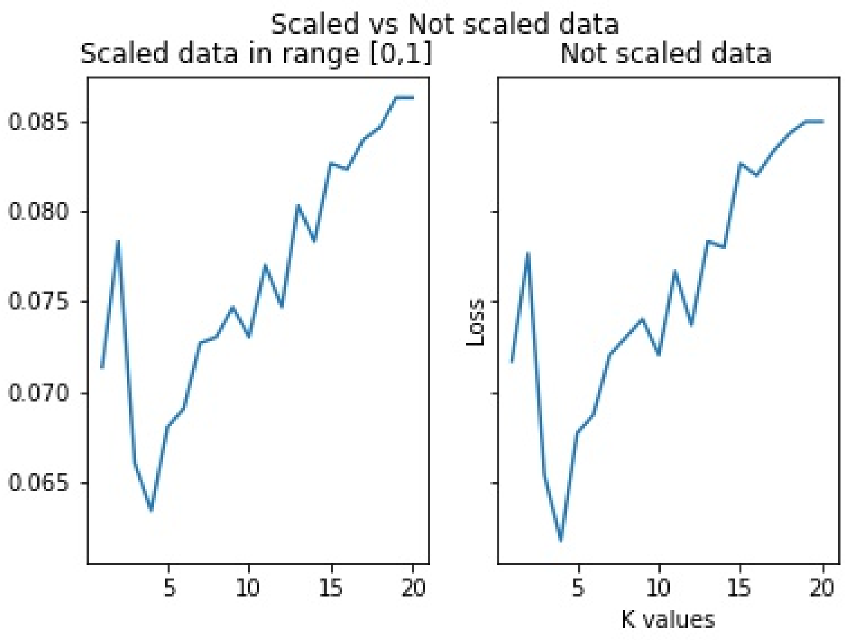


Figure 5: Scaled data in [0,1] vs not scaled data

definition is shown below:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

For the image blurring technique, instead, we used the 5×5 Gaussian Blurring kernel. It is a widely used technique to reduce noise and details in images.

$$K = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

In the image below we can see the transformation applied by these kernel on a sample image. Also, it is shows the loss for different k and different prepossessing techniques, on the KNN with Mikowski distance with $p = 11$.

In order to have a benchmark, we use cross-validation on training data for different k and plot the loss resulted in order to get a better understanding of the role played by the preprocessing. Overall, we cannot see an improvement with the blurring or the averaging. However, we assume that some other techniques may have a better result but, since the computation of the loocv required a relevant time, we did not delve further. The optimal k, in this scenario, is $k = 4$ and with no preprocessing.

Even though these approaches can have a positive effect, the algorithm will still have to cope with a high number of features that characterized each image. To face this problem we decided to reduce the number of features of each image from $28 \times 28 = 784$ to $14 \times 14 = 196$. To do so, we decided to merge groups of four features (pixel) into one by computing the average of these. In that way, we lose some details of the information, but we also reduce the size of each data point by a factor of 4. This indeed allows the algorithm to be faster and it still ensures the learning capability of the classifier. In Figure 8 it is possible to see the same trend of the loss given by the leave-one-out cross-validation for the different dataset considered. For

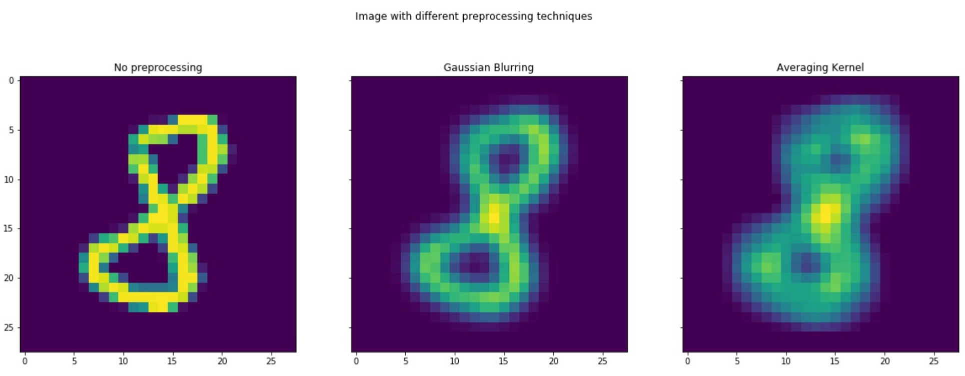


Figure 6: Loss heatmap for various $p \in \{1, \dots, 15\}$ and $k \in \{1, \dots, 20\}$

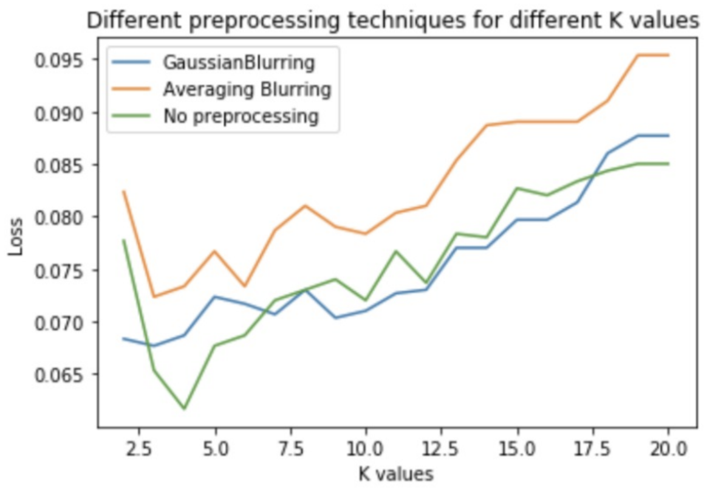


Figure 7: LOOCV with different k neighbors and preprocessing techniques

the original dataset where each image is composed of 784 pixels the time required to fit the algorithm was 46.31 sec, while for the dataset where the images have been reduced the fitting time is just 12.11 sec.

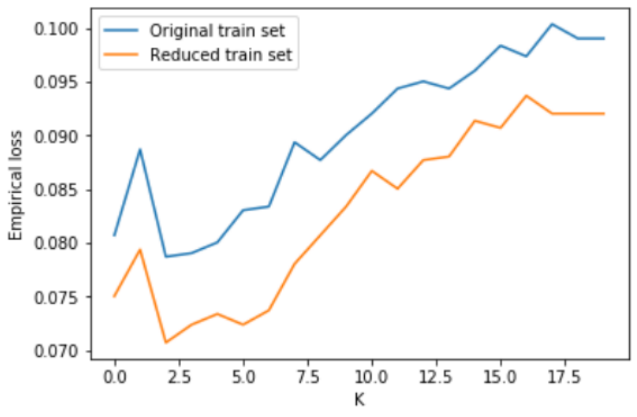


Figure 8: Empirical loss for train set using images of 784 pixels and 196 pixels

3 k-NN implementation with big MNIST dataset

3.1 Risk estimation via cross-validation

The best performance of our k-NN implementation for the small dataset was using the Minkowski distance with $p = 11$ and $k = 4$. The next step is to test how these methods scale to the full dataset. However, our program was not ready for the dimensions of this one. We had both memory and computational issues.

To cope with a large amount of data we implemented two changes.

- We stored a matrix with the distances between all the points.

Since for the training process we have to compute the distance between each point, is it not smart to do it iteratively. Doing so, the program will duplicate computations because $d(a, b) = d(b, a)$ for each a, b. Hence, what we first did was to compute and store a $n \times n$ matrix with the distance values. Then, we would access that matrix during the training process to select the k nearest neighbors.

Even though this optimization reduced computational time, we had to store a huge matrix, so the program was not yet really scalable for the big dataset. Thus, we implemented a second change.

- We stored only the indexed of the first 20 neighbors for each point.

We realized that the distance matrix contained many distances that might not be necessary for the future. Since only the k-nearest neighbors are involved, we transformed that matrix to a $n \times 20$ matrix containing, for each point, the indices of the 20 nearest points. We chose 20 because is the higher k we will test.

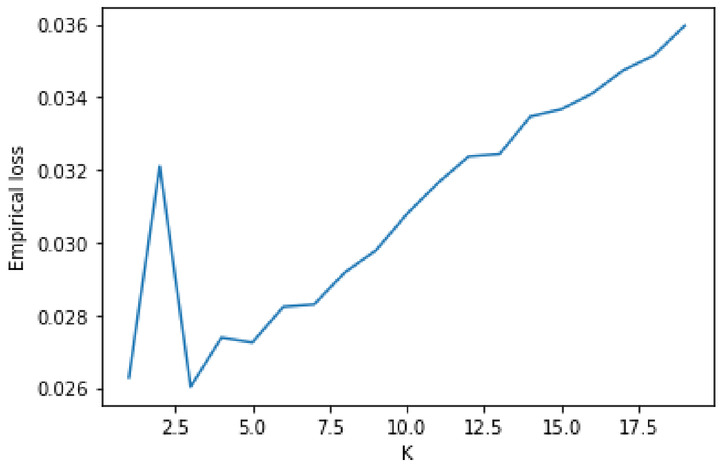


Figure 9: Empirical loss for train set using $p = 11$ for $k \in \{1, \ldots, 20\}$

In Figure 9, we see that after reaching the minimum at $k = 3$ there is an increase in the empirical loss as the value of k grows. As the number of neighbors in consideration increases, it increases the number of data points from many different classes (that are relatively distant) which affects the voting for the class and hence the performance of the classifier.

3.2 Determining empirical loss

The empirical loss with the test data is more than that with the training data for each value of k. The performance of the smaller training dataset for Minkowski distance (with $p = 11$) metric is better than that of the larger training dataset for smaller number of neighbors. While there is an increase in empirical loss from 0 to 0.08, the values remain between 0.05 to 0.08 for larger training data.

Moreover, it is interesting to note that the trend with the test data is almost the same as that of the training data. We would choose $k = 2$ had we been allowed to see the test data set. The empirical loss for the test data set is almost the same for $k = 6$ as well. However, $k = 2$ gives the best result for both the data sets.

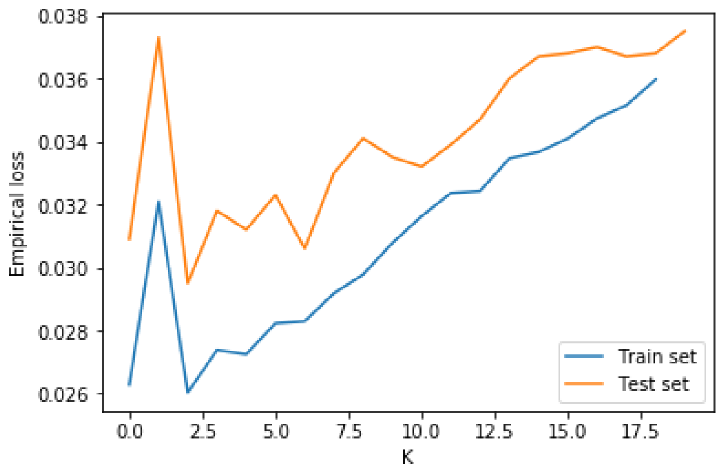


Figure 10: Empirical loss for train and test set using $p = 11$ for $k \in \{1, \ldots, 20\}$

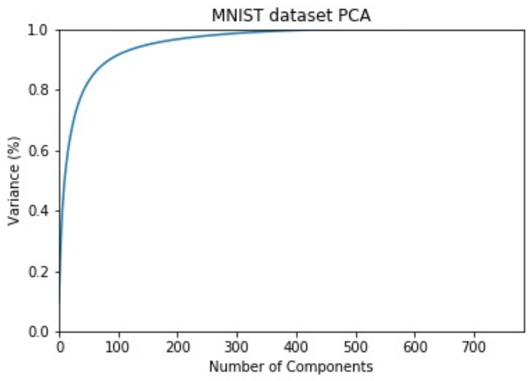
4 PCA

Every single image is made of 784 pixels and we must point out that the KNN algorithm is substantially influenced by the size of the training dataset. Specifically, in our simplest implementation approach, by using a brute force distance matrix we need to compute (for the big dataset) a 60000×60000 matrix and this is very inefficient in terms of memory and speed.

By using some kind of dimensionality reduction, it is possible to preserve the most important features for each observation and therefore reducing the size of the training dataset. Besides the computational advantage in computing distances, there might be also an advantage in terms of statistical generalization.

In our case, we used the **Principal Component Analysis (PCA)** technique, which converts a set of observations into linearly uncorrelated variables so that they can capture most of the variability in the data. We must point out that PCA is sensitive to scaling and therefore we address this problem by first apply normalization to the data and reduce the variability of each point in the range [0,1]. We used the implementation provided by the python library sklearn to extract the principal components. In order to choose the number of principal components, we first decided to plot the cumulative variance expressed by these variables so it is possible to arbitrarily pick a number that expresses high variability in the data. As it is possible to see in figure 11, already less than 100 features can express more than 80% of the variability of the dataset. We hence zoomed in the range $[0, 100]$ and decided to pick initially 50 principal components (considering that our dataset is big and we want to speed up the training). After fitting the training dataset and compute the score on it, we can decide to further increase the number of principal components or vice versa.

We evaluated our new input data with the Leave One Out cross-validation. As before, we use Mikowski distance with $p = 11$ and considered the first $k = 4$ nearest neighbors. In this case the computational time has been drastically reduced and we can still obtain an accuracy score equal to 0.9705.



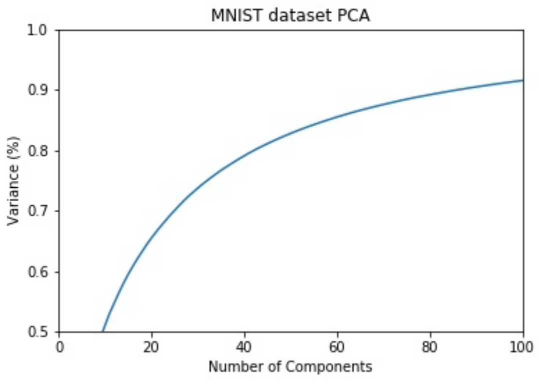


Figure 11: Variability expressed by different number of principal components. In figure on the right the range is narrowed down to $[0, 20]$