Statistical Learning Theory - 2DI70

Part I - Computational Assignment

Nearest Neighbor Classification and Handwritten Digit Classification

Group 10:
Jelle Ster - 0961117
Tos Kuipers - 0679808
Huyen Nguyen - 1319531

Eindhoven, 13 March 2019

Overview

The first assignment is to classify handwritten digit of numbers and to predict its label from 0 to 9. Here, we will use MNIST dataset and compute k-Nearest Neighbor (k-NN) method using Euclidean distance and Minkowski distance approach. The k-value of k-NN (ranged from 1 - 20) will be reviewed in each distance approach to discover the best value of k. The report will give you modifications for improving performance of k-NN method as well Principal Component Analysis (PCA) settings as a part of improving the performance. A small dataset of training set (3000 samples) and test set (1000 samples) will be used to work in part 1a till 1d, while part 1e till 1g will be used with the full MNIST dataset of training set (60000 samples) and test set (10000 samples).

For the remaining of the report, please do notice that we programmed the code for calculation in a manner which skips the first index of the data set. Thus, iteration on the small training set will go over 2999 data points, iteration on the small test set will go over 999 data points and so on.

1a. Implementation of k-Nearest Neighbors (k-NN)

In this part we will build the k-NN method with Euclidean distance approach and report on its accuracy to predict the labels in both training and test sets. As introduced earlier for this part, only small sets will be used to work on. An overview of steps to implement the k-NN method[1]:

[1] https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/ Accessed in March 2019.

Step 1. Calculate the Euclidean distance between each selected data point to predict and all data points within the training set. The selected data point to predict can be either data in the training set itself or data in the test set.

$$\text{Euclidean distance} = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

$$where\ d = number\ of\ selected\ data\ points\ to\ predict,$$
$$x = selected\ data\ point\ to\ predict,\ y = data\ point\ in\ training\ set$$

Step 2. Sort the calculated distances in ascending order of distance values.

Step 3. Get top k rows from the sorted array (k can range from 1 - 20).

Step 4. Get the most frequent class of these rows (the class is the label of the training data point with least distance to the selected data point and with the most frequency).

Step 5. Return the predicted class as the most frequent class of the rows.

Step 6. Compare the predicted class with the actual class. Since the selected data point to predict can be either the training data or the test data, here we will need to compare the predicted class with label in the training or test data set according to what we use as the selected data.

We use a loss function of 0/1 such that if the predicted class is different from the actual class, then the loss function will get a value of 1, otherwise 0. Summing up all the values of the loss function and dividing by number of selected data points, we obtain empirical risk value:

$$\frac{1}{m}\sum_{i=1}^{m} 1\{\widehat{Y}_i \neq Y_i\},\ where\ \widehat{Y}\ is\ predicted\ class,\ Y\ is\ actual\ class.$$

Table 1 contains a summary of risk estimates on both training and test set over different k from 1 to 20, while figure 1 shows the graphical view of the empirical risk.

k-value	Risk estimates on training set	Risk estimates on test set
1	0.00	0.0850850850850850
2	0.00	0.0850850850850850
3	0.0273424474824941	0.0770770770770770
4	0.0270090030010003	0.0760760760760760
5	0.0406802267422474	0.0720720720720720
6	0.0426808936312104	0.0760760760760760
7	0.0513504501500500	0.0800800800800800
8	0.0533511170390130	0.0810810810810810
9	0.0623541180393464	0.0830830830830830
10	0.0626875625208402	0.0900900900900900
11	0.0706902300766922	0.0950950950950950
12	0.0703567855951984	0.0970970970970971
13	0.0753584528176058	0.0980980980980981
14	0.0733577859286428	0.1031031031031030
15	0.0803601200400133	0.1031031031031030
16	0.0806935645215071	0.1051051051051050
17	0.0843614538179393	0.1041041041041040
18	0.0830276758919639	0.1111111111111110
19	0.0843614538179393	0.1121121121121120
20	0.0853617872624208	0.1131131131131130

Table 1: Risk estimates on k-NN in both training and test set

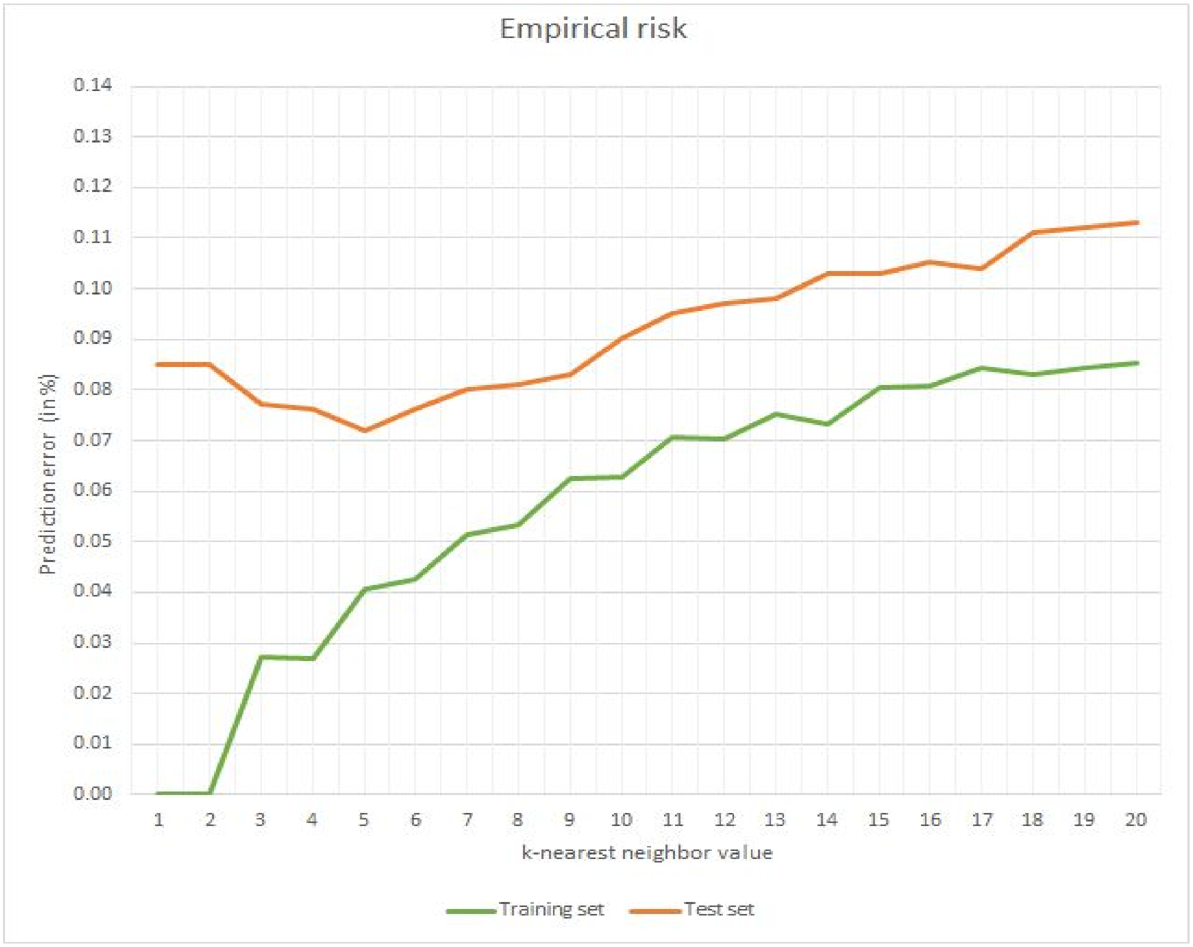


Figure 1: Empirical risk on k-NN in both training and test set

According to the table and the figure above, the training set has better performance than the test set. Accuracy in the training set is on average 3% higher than in the test set.

For the training set, with k = 1 and k = 2, the accuracy was 100% correct. Since we use data within the training set itself to predict label, the nearest neighbor of distance 0 will be existing as the data point itself. Thus, we obtain an accuracy of 100%, however, those k-values overpower the prediction. With k increasing from 3 to 20, the prediction error percentage increases accordingly.

For the test set, initial k = 1 and k = 2 returns a high prediction error of about 8.5%, then from k = 3 to k = 6 the error percentage decreases slowly to 7.6%. After that the error prediction rises up again.

1b. Implementation of Leave-One-Out-Cross-Validation (LOOCV)

In this part, we will implement the k-NN method using Leave-One-Out-Cross-Validation (LOOCV) approach. LOOCV is a member of k-fold cross validation family in which "*the data set is divided into k subsets [...] Each time, one of the k subsets is used as the test set and the other k-1 subsets are put together to form a training set. Then the average error across all k trials is computed*". [2] Below figure 2 shows how LOOCV with k-folds works.

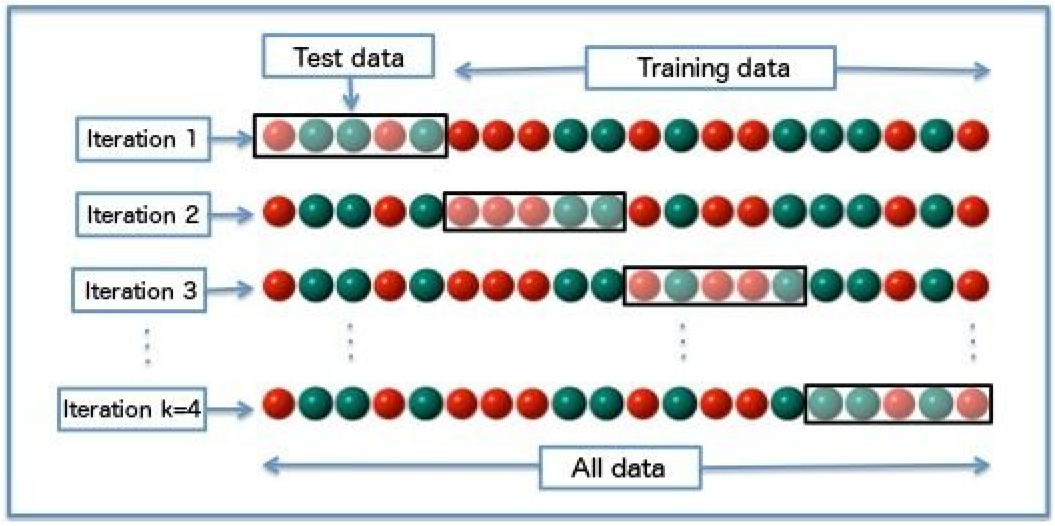


Figure 2: Illustration of k-folds cross validation

LOOCV is a "*K-fold cross validation taken to its logical extreme, with K equal to N, the number of data points in the set. That means that N separate times, the function approximator is trained on all the data except for one point and a prediction is made for that point*".[2] Each data point in the training set will be held out to get distance with all other data points in the training set (except itself). Thus the name of LOOCV was used. The implementation of k-NN with LOOCV approach remains mainly the same as the original k-NN in part 1a. However, when calling for the collection set of training set, we will use the collection of all data points in the training set excluding the selected data point to predict. Every data points gets predicted once, and the average error is computed using the 1/0 loss rule. Table 2 below will give you a summary of risk estimates using LOOCV approach on k-NN method. Only the training data set is used here.

[2] https://www.cs.cmu.edu/~schneide/tut5/node42.html Accessed in March 2019.

1b - LOOCV Risk Estimates for $k \in \{1, ..., 20\}$			
k-value	**LOOCV Risk Est.**	**k-value**	**LOOCV Risk Est.**
1	0.08069356452150717	11	0.09203067689229744
2	0.08869623207735912	12	0.09436478826275425
3	0.07869289763254418	13	0.09503167722574192
4	0.07902634211403801	14	0.09436478826275425
5	0.0800266755585195	15	0.0960320106702234
6	0.08302767589196398	16	0.09836612204068022
7	0.08336112037345782	17	0.09736578859619874
8	0.08936312104034679	18	0.10036678892964321
9	0.08769589863287762	19	0.09903301100366789
10	0.09003001000333445	20	0.10103367789263087

Table 2: Risk estimates on k-NN using LOOCV approach

Below, figure 3 contains an overview of empirical risk for training set using LOOCV approach together with empirical risk for test set in part 1a.

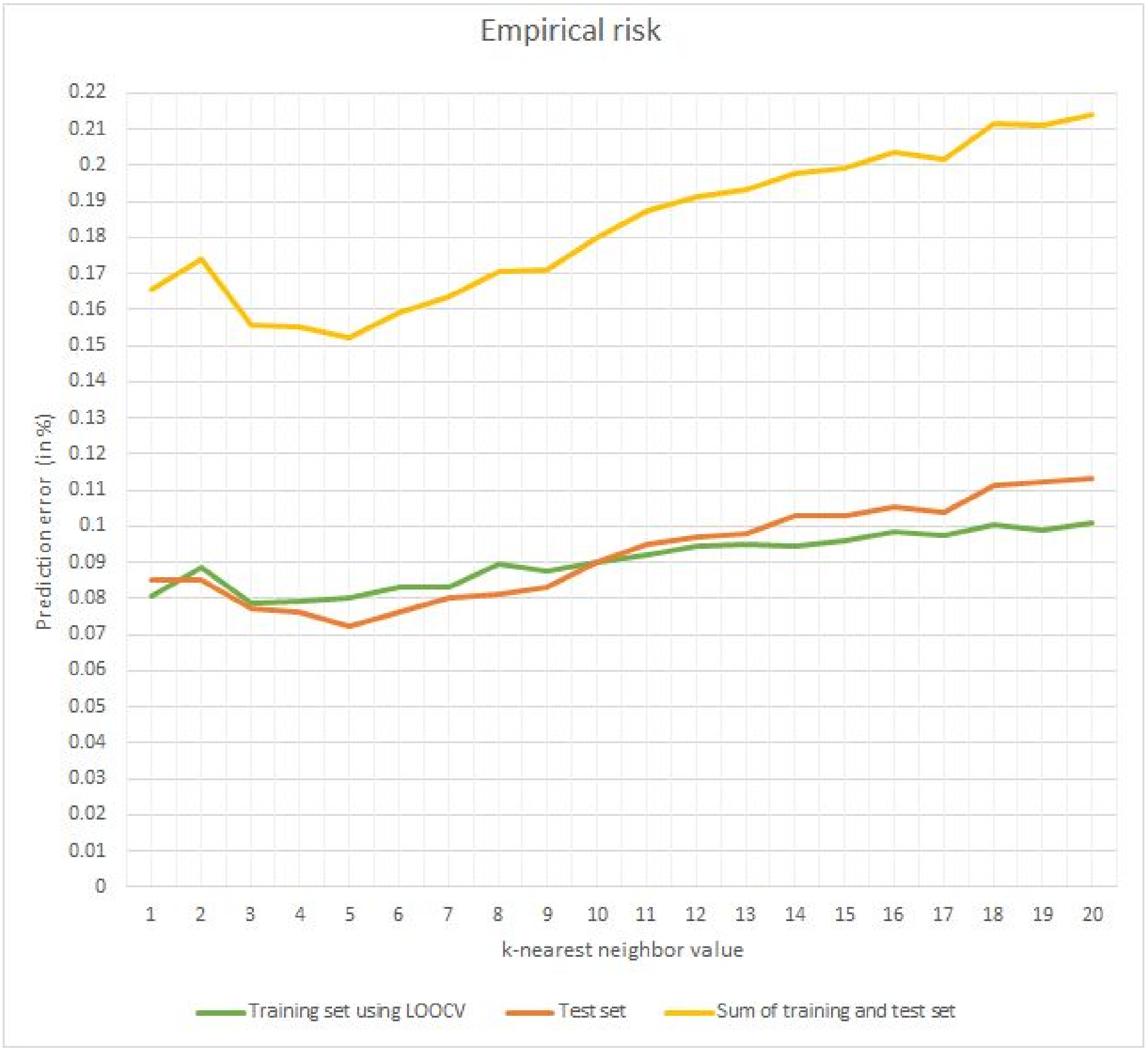


Figure 3: Empirical risk on k-NN in training set (using LOOCV) and in test set

Overall, per training set and per test set, the risk gets higher when k gets higher. Given the results in figure 3, a good choice of k will be 5 since after summing up both the training and test set's risk value, at k = 5, the total risk gets its lowest point. Though, without looking at the test set, k = 3 would be the optimal choice.

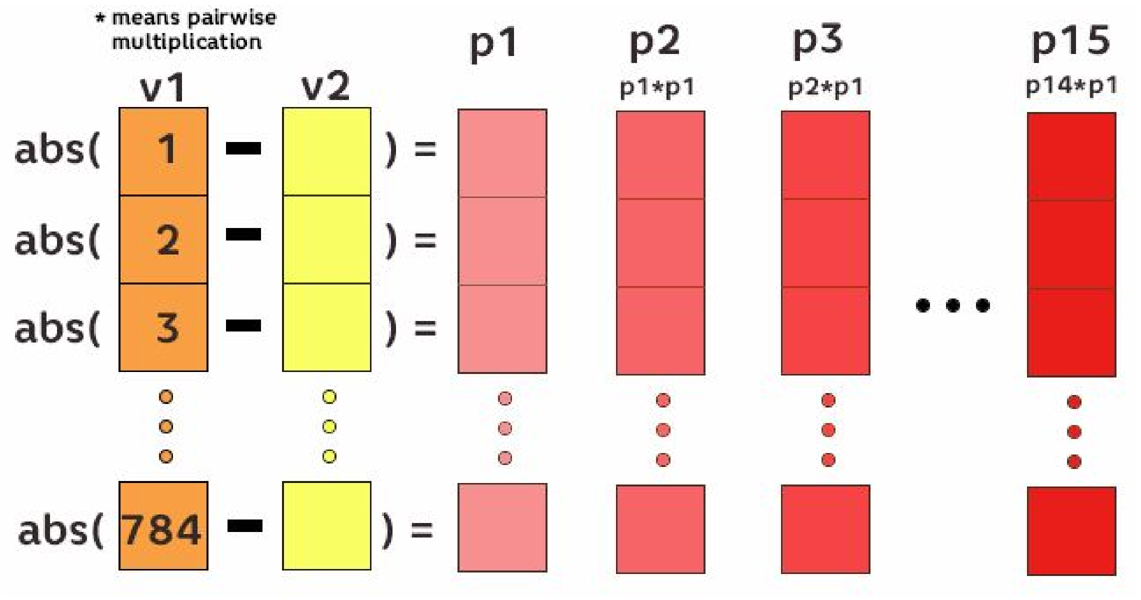
1c. Implementation of LOOCV with Minkowski distance

Minkowski distance is given by the formula: $(\sum_{i=1}^{d} |x_i - y_i|^p)^{1/p}$, where x = the selected data point to predict, and y is a data point from the training set, with p≥ 1 .

For the work in this report, we will use p value from 1 to 15, with p is integer. Even though the p-value in Minkowski formula can be real number; any value in between the range [1,2], [2,3], [3,4] and so on, are limited by the lower bound and upper bound of the integer value. Thus knowing the value of integers can inform us the range of real value happening in between the integer. In order to determine both a suitable k and p at the same time, we need to perform a grid search with LOOCV. This requires many calculations. We therefore introduce a number of speedups into our algorithm.

Recursive Minkowski distance calculation

Because we already know we will need to calculate 15 Minkowski distances between each two points of data in our training set, we can do them all at once at the start, instead of once every iteration of the grid search. The advantage of this is that we can use previous distances to speed up calculations. This works as follows: When calculating a Minkowski distance between two arrays (say **v1** and **v2**), we first have to find the difference between all pairs of points. We will call this array **p1.** The values of the resulting array are then raised to the power p. It is in this step we can improve performance by recursion. Instead of raising to the power of p, we can simply take the array of a previous result and multiply the values with the values in **p1**. After we



have all 15, we can then sum the values in each of them and compute the p-th root of each of those sum. This significantly reduces the number of operations we need to perform.

Reusing distances

The fact that distance is symmetric can be used to further increase performance. Instead of re-calculating distances between two data points **a** and **b**, we can save the distance when we first encounter it. Then, when we later need to calculate the distance from **b** to **a**, we already have the result. This roughly halves the number of operations we need to perform.

Even with these performance improvements, the sheer number of calculations still takes about 6 hours. It shows one of the weaknesses of knn; performance drops drastically when the size of the dataset scales up.

Results

Results are shown in the table below. Higher values means higher average error. We can see the best performance lies at *p=11* and *k=4*.

Average Prediction Error of LOOCV on training set for k from 1 to 20 and p from 1 to 15

P K	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0.090	0.100	0.094	0.092	0.096	0.096	0.098	0.097	0.100	0.103	0.102	0.103	0.105	0.106	0.110	0.111	0.114	0.116	0.118	0.120
2	0.081	0.089	0.079	0.079	0.080	0.083	0.083	0.089	0.088	0.090	0.092	0.094	0.095	0.094	0.096	0.098	0.097	0.100	0.099	0.101
3	0.075	0.082	0.071	0.074	0.075	0.080	0.080	0.082	0.084	0.084	0.087	0.090	0.089	0.088	0.092	0.092	0.095	0.095	0.094	0.098
4	0.073	0.076	0.070	0.072	0.072	0.077	0.077	0.079	0.080	0.079	0.079	0.084	0.084	0.086	0.089	0.089	0.091	0.092	0.092	0.094
5	0.073	0.075	0.065	0.071	0.068	0.074	0.075	0.078	0.079	0.077	0.079	0.079	0.081	0.083	0.083	0.084	0.090	0.090	0.087	0.093
6	0.071	0.074	0.065	0.066	0.069	0.072	0.074	0.076	0.078	0.076	0.076	0.077	0.081	0.081	0.084	0.083	0.085	0.086	0.086	0.088
7	0.071	0.075	0.063	0.065	0.069	0.073	0.072	0.075	0.078	0.075	0.078	0.075	0.080	0.081	0.082	0.083	0.085	0.083	0.084	0.087
8	0.069	0.076	0.063	0.065	0.068	0.072	0.072	0.076	0.076	0.072	0.076	0.075	0.080	0.080	0.081	0.082	0.084	0.083	0.086	0.088
9	0.069	0.076	0.063	0.063	0.068	0.072	0.073	0.074	0.074	0.072	0.078	0.075	0.079	0.079	0.080	0.084	0.084	0.083	0.086	0.087
10	0.070	0.076	0.065	0.063	0.067	0.069	0.072	0.072	0.073	0.072	0.079	0.075	0.078	0.078	0.081	0.083	0.083	0.083	0.086	0.087
11	0.072	0.078	0.065	0.062	0.068	0.069	0.072	0.073	0.074	0.072	0.077	0.074	0.078	0.078	0.083	0.082	0.083	0.084	0.085	0.087
12	0.071	0.077	0.066	0.063	0.067	0.070	0.073	0.072	0.073	0.072	0.075	0.073	0.078	0.077	0.082	0.080	0.084	0.083	0.086	0.087
13	0.070	0.077	0.068	0.063	0.067	0.070	0.073	0.071	0.073	0.072	0.076	0.073	0.077	0.076	0.080	0.080	0.083	0.083	0.084	0.087
14	0.071	0.078	0.069	0.064	0.068	0.069	0.072	0.071	0.071	0.073	0.074	0.071	0.076	0.076	0.079	0.080	0.083	0.084	0.085	0.087
15	0.070	0.079	0.069	0.064	0.068	0.069	0.072	0.070	0.071	0.073	0.074	0.071	0.076	0.076	0.080	0.081	0.084	0.084	0.086	0.086

Note that the results we find here for *p=2* concur with those we found in 1b.

1d. Adjustment for improving Nearest Neighbors performance

The MNIST dataset contains handwritten digits and their corresponding label. Differences between handwritten digits can be rather small, as can be seen in the training data, where entries with similar labels can have slightly contrasting digits. As such, it is worth considering a change to the distance metric that is more sensitive to smaller changes. An example of this includes using the natural logarithm, or $ln(x)$, of the difference between two vectors to predict the label. This distance metric is known as the Lorentzian distance (LD)[3].

Since the natural log of 0 is undefined, the LD prevents reaching this situation by adding 1 to the absolute vector changes. Thus, the Lorentzian distance metric is defined as follows:

$$d_p(x, y) = (\sum_{i=1}^{d} ln(|x_i - y_i| + 1)^p)^{1/p}$$

1e. LOOCV method with Euclidean/ Minkowski distance on full training dataset

Our implementation, making use of either the Euclidean distance or d_{11} according to the results of 1c, was unable to cope with the full training dataset of 60.000 examples. The running time of the LOOCV method to determine a good value for k would effortlessly exceed ten hours per considered value of k, which we deemed impractical.

1f. LOOCV method with Euclidean/ Minkowski distance on full test dataset

Because our implementation could not cope with the large amount of data of the full training dataset, we only attempted to compute the loss of our method on a subset of the first 500 elements from the test set, using the Euclidean distance, $p = 2$, and the value for k obtained during 1b, $k = 3$, but due to time constraints this did not lead to results either.

[3] V.B.S. Prasath et al. (2017). Distance and Similarity Measures Effect on the Performance of K-Nearest Neighbor Classifier - A Review.

1g. Implementation of Principal Component Analysis (PCA)

The current dataset contains 28*28=784 attributes/dimensions for each data point/sample. Having many attributes may contribute to overfitting the model to predict the label for the data. Not all the dimensions are relevant to the generalization capacity of the model.[4] Reducing number of dimensions into smaller subspace can lower the computational cost of the model and the error of parameter estimation. PCA is the analysis used to identify and find patterns to "reduce the dimensions of the dataset with minimal loss of information", so that the created subspace with reduced dimensions can remain to represent the dataset well.[5]

For the problem of classifying handwritten digits to number from 0 till 9, we apply the concept of PCA using ready implementation of PCA from scikit-learn and Numpy libraries of Python. The PCA class has its "explained variance" analysis for each attribute, giving us the number of components/ dimensions, and how much preservation in percentage we can obtain in the total variance of the data. Figure 4 below contains a plot of the explained variance analysis on the full training set of MNIST.

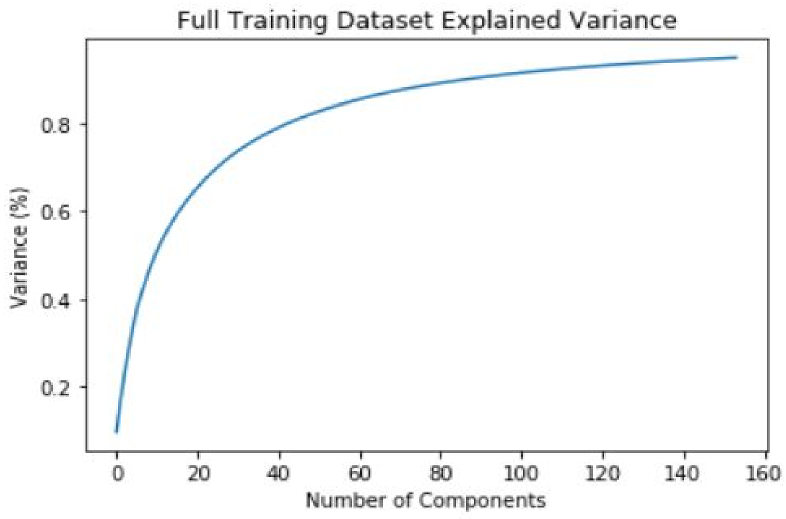


Figure 4: Explained variance analysis on the full training set of MNIST

[4] https://towardsdatascience.com/an-approach-to-choosing-the-number-of-components-in-a-principal-component-analysis-pca-3b9f3d6e73fe Accessed in March 2019

[5] https://sebastianraschka.com/Articles/2014_pca_step_by_step.html#6-transforming-the-samples-onto-the-new-subspace Accessed in March 2019

100% of variance denotes the use of all components. Since we only want to use principal components, having 95% is an optimal choice for remaining data information while reducing dimensions[6]. As the plot has shown you, in order to obtain 95% of the total variance of the data, we need to use about 150-160 components. Calling for a precise number of components/ dimensions in the PCA class, we get a value of 154 dimensions.

After obtaining the number of reduced dimensions, we will fit the PCA model to the dataset and then transform the dataset to a reduced-dimension dataset of only 154 dimensions. The reduced-dimension dataset (PCA dataset) will then be applied when running the k-NN method. Thus, it is the same procedure as the k-NN implemented in part 1a or suggested in 1d. However, when setting the parameter for the training set, we will use the PCA dataset instead. As such, the distance between the selected data point and the training set will be replaced by the distance between the selected data point and the PCA dataset.

Even though the dimensions are reduced, using the full MNIST training dataset of 60000 samples under our implemented k-NN model is not sustainable. The time taken to complete the run on the full training set is approximately 40 hours.

In order to discover how PCA can boost up the procedure of k-NN model, we apply the PCA on the small set of training data. It shows that the performance (in term of speed) with PCA on k-NN is better than only using k-NN at least 7 times. The time to generate a label prediction and comparing with the actual label in k-NN takes about 7 seconds, while with PCA it takes less than a second. In terms of accuracy, with PCA applied, the risk estimate is slightly lower than the risk estimate using original k-NN, as shown in 1a. In the figure 5 below, a comparison of the risk estimates using k-NN (part 1a), and using PCA to the k-NN, is shown.

[6] https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60 Accessed in March 2019

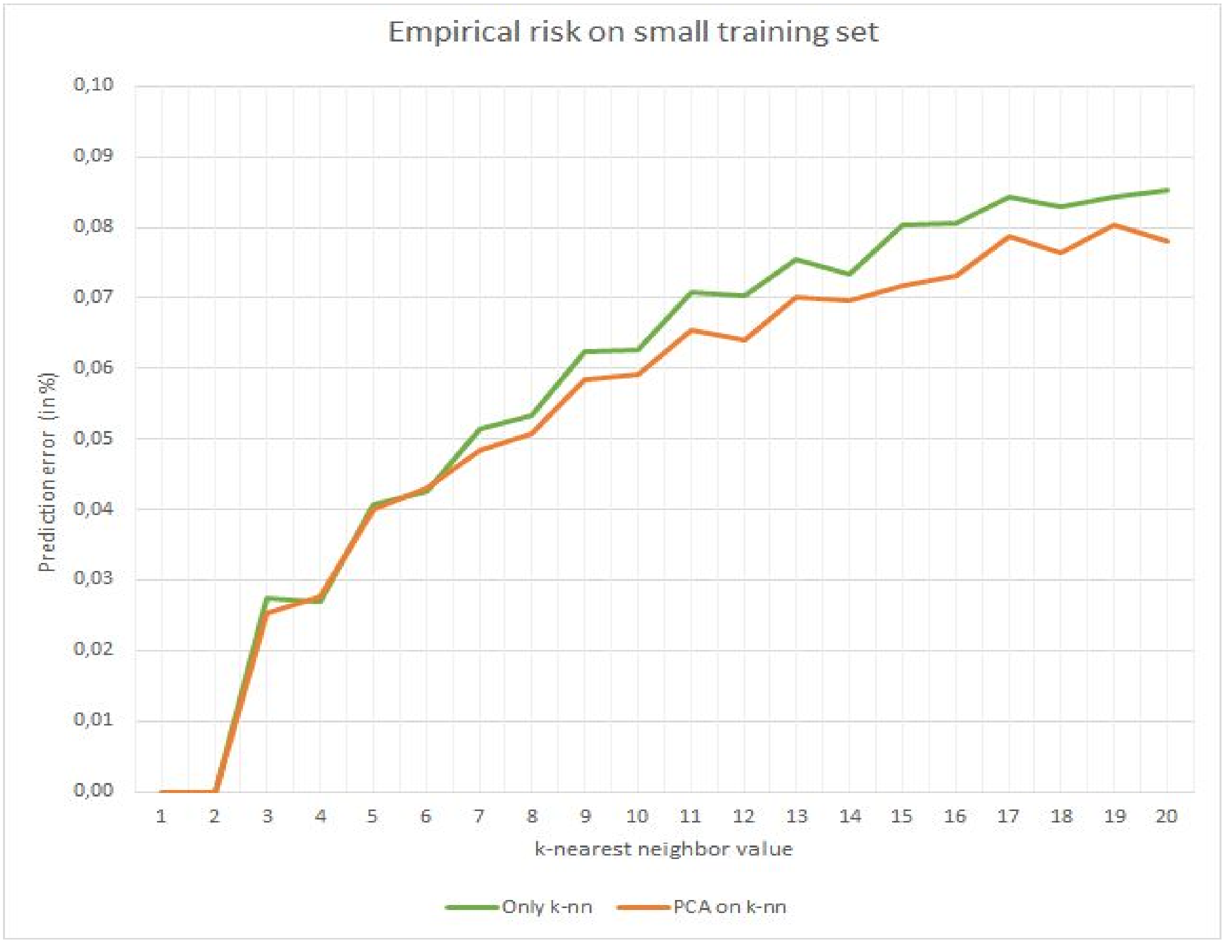


Figure 5: Comparison of k-nn with and without PCA