Exploring Classification Methods for Fisher´s Iris Dataset and The MNIST Dataset of Handwritten Digits

Group 22
Håkon Hjortdahl, 564640
Rasmus A. F. Nordal, 564647

April 30, 2024

NTNU | Department of Electronic Systems

Abstract

This report examines the performance of a linear classifier for classification of Fisher's Iris dataset, and a k-NN classifier – with and without K-Means clustered templates – for classification in the MNIST dataset. We proceed to discuss properties like correlation and separability in the Fisher's Iris dataset. The dataset was found to be nearly linearly separable, allowing the linear classifier to perform well, even when reducing the number of features. With all features, test set error rates ERR_T of 1.67% and 3.33% were reached for two different partitionings of training and test data. In the MNIST task, an NN classifier without clustering gave the lowest error rate, with $ERR_T = 3.09\%$, while using approximately 26 minutes to classify all 10000 test samples. Clustering templates to 64 cluster centers per class increased ERR_T to 4.7% and 5.47% with NN and 7-NN respectively, while reducing processing time to approximately 37 seconds in both cases.

Contents

1 Introduction

Classification algorithms are of paramount importance in today's society, with applications ranging from medical diagnostics to signal processing to financial markets.

The goal of this project is to examine a set of classical classification methods, apply them to two classification problems, and in the process broaden our understanding of the methods. The two problems consist of classification of the famous Fisher's Iris dataset and MNIST dataset. Our report is divided into two main sections, each addressing one of the classification problems. Each section includes a discussion of the relevant theory and the classification methods used, an account of our implementation, and a presentation and discussion of our results. The report is then summarized in a joint conclusion.

2 Classification of Fisher's Iris Dataset

2.1 Theory

Fisher's Iris dataset consists of measurements of 50 specimen each of three different variations of the iris flower; the *Iris Setosa*, *Iris Versicolor* and *Iris Virginica*. The measurements, or features, are the length and width of the sepal and petal leaves. The accompanying classification problem consists of classifying the variation based on the features. This dataset is well known for being one of few real-world datasets that are close to linearly separable, meaning it can be successfully classified using only hyperplanes as decision boundaries [2]. Therefore, a linear classifier is used for this task.

A linear classifier is a supervised learning algorithm, meaning that the method uses labeled data to classify new data into known classes. For C classes, each class ω_c has a corresponding discriminant function $g_c(\boldsymbol{x})$, and the predicted class $\hat{\omega}$ is calculated from the decision rule

$$\hat{\omega} = \omega_{c^*}, \text{ where } c^* = \arg\max_c g_c(\boldsymbol{x}). \tag{1}$$

Each discriminant function is a transformation of a weighted sum of the features plus a constant bias term. Representing the features on homogeneous form, $\tilde{\boldsymbol{x}}^T = \left(1, \boldsymbol{x}^T\right)$, we can write the weighted sums plus bias terms as the matrix product

$$\boldsymbol{z} = \boldsymbol{W}\tilde{\boldsymbol{x}}, \tag{2}$$

where $\boldsymbol{W}$ is referred to as the classifier's weight matrix. The mentioned transformation is the sigmoid function, which can be interpreted as a differentiable approximation of the Heaviside step function. It is defined as

$$\boldsymbol{g} = \boldsymbol{\sigma}(\boldsymbol{z}) \Leftrightarrow g_c = \frac{1}{1+e^{-z_c}}, \quad c = 1, ..., C. \tag{3}$$

Altogether, this gives the following vector of discriminant functions;

$$\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{\sigma}(\boldsymbol{W}\tilde{\boldsymbol{x}}) \tag{4}$$

The classifier is then trained by minimizing an optimization criterion over $\boldsymbol{W}$. In our case, the criterion is the Mean Square Error (MSE), which for N samples is defined as

$$\text{MSE} = \frac{1}{2}\sum_{k=1}^{N}(\boldsymbol{g}_k - \boldsymbol{t}_k)^T(\boldsymbol{g}_k - \boldsymbol{t}_k), \tag{5}$$

where $\boldsymbol{t}_k$ is a one-hot encoding of sample k's class ω_k. Optimization is done by gradient descent, where for every L samples (referred to as one batch), $\boldsymbol{W}$ is modified in the opposite direction of the gradient

$$\nabla_{\boldsymbol{W}}\text{MSE} \approx \sum_{k=1}^{L} \nabla_{\boldsymbol{g}_k}\text{MSE}\,\nabla_{\boldsymbol{z}_k}\boldsymbol{g}_k\,\nabla_{\boldsymbol{W}}\boldsymbol{z}_k. \tag{6}$$

Some simple differentiation yields

$$\begin{aligned}\nabla_{\boldsymbol{g}_k}\text{MSE} &= \boldsymbol{g}_k - \boldsymbol{t}_k \\ \nabla_{\boldsymbol{z}_k}\boldsymbol{g}_k &= \boldsymbol{g}_k \circ (1 - \boldsymbol{g}_k) \\ \nabla_{\boldsymbol{W}}\boldsymbol{z}_k &= \boldsymbol{x}_k^T,\end{aligned} \tag{7}$$

giving

$$\nabla_{\boldsymbol{W}}\text{MSE} \approx \sum_{k=1}^{L}\left[(\boldsymbol{g}_k - \boldsymbol{t}_k) \circ \boldsymbol{g}_k \circ (1 - \boldsymbol{g}_k)\right]\boldsymbol{x}_k^T \tag{8}$$

$\boldsymbol{W}$ is then updated by

$$\boldsymbol{W}(m) = \boldsymbol{W}(m-1) - \alpha\nabla_{\boldsymbol{W}}\text{MSE}, \tag{9}$$

where α is the step size, which is tuned through trial and error.

In any dataset, each feature will contain some portion of the information that can be used to classify the samples. In some datasets, a given feature may add little information while increasing model complexity and the risk of overfitting. Overfitting is when the model adapts closely to the training set, instead of learning the general trends that also apply to unseen data. Therefore, a classifier may benefit from removing a feature from the dataset.

When evaluating the performance of a classifier, a common metric is its error rate p_e. In most practical cases, where the exact distribution of the data is not known, the true error rate of a classifier can only be estimated from the data. Furthermore, classification of data can be thought of as a Bernoulli process with equal probability of error p_e for each sample. It is well known that the maximum likelihood estimator (MLE) of the error rate in a Bernoulli process is

$$\hat{p}_e = E/N \text{ with } \operatorname{Var}\{\hat{p}_e\} = \frac{p_e(1-p_e)}{N}, \tag{10}$$

where E and N represent the number of errors and trials, respectively. For a low N, such as in Fisher's Iris Dataset, this variance has a large impact. To represent it, a confidence interval (CI) is well estimated by a Wilson score interval [7], given by

$$\Pr\left\{w^- \leq \hat{p}_e \leq w^+\right\} \approx \alpha, \tag{11}$$

where α is the significance level and

$$w = \frac{E + \frac{z^2_{\alpha/2}}{2} \pm \sqrt{\frac{E(1-E)}{N} + \frac{z^2_{\alpha/2}}{4}}}{N + z^2_{\alpha/2}}. \tag{12}$$

An important detail is the distinction between the training and test dataset; Only the training set is used to tune the model's parameters, while the test set is used to estimate its performance on unseen data. Therefore, the test error rate ERR_T is used to estimate the true error rate:

$$\hat{p}_e = ERR_T = \frac{E_T}{N_T}, \tag{13}$$

where E_T and N_T represent the number of misclassified test samples and total test samples, respectively.

The training error rate ERR_D is also a useful metric, as it may indicate how well the model generalizes; If ERR_T is notably greater than ERR_D, it may be a sign of an overfitted model [2].

2.2 Implementation

A linear classifier was implemented as described in Section 2.1. Python code is given in Appendix B. As the dataset is rather small, the batch size was set to equal the size of the training set, i.e. $L = N$.

To determine a suitable step size α, multiple values were attempted, giving the results in Fig. 1. $\alpha = 0.1$ does not converge at all, and $\alpha = 0.01$ shows an almost oscillating behaviour, possibly because a larger step size is more likely to step over candidate minima. $\alpha = 0.005$ seemed to be a suitable value, so this was chosen for all subsequent experiments. With this step size, the classifier converges within 1500 training steps, so this number of steps is also used in further experiments.

In addition to choosing the step-size, the training and validation sets were chosen. Two variations were examined; Using the first 30 samples of

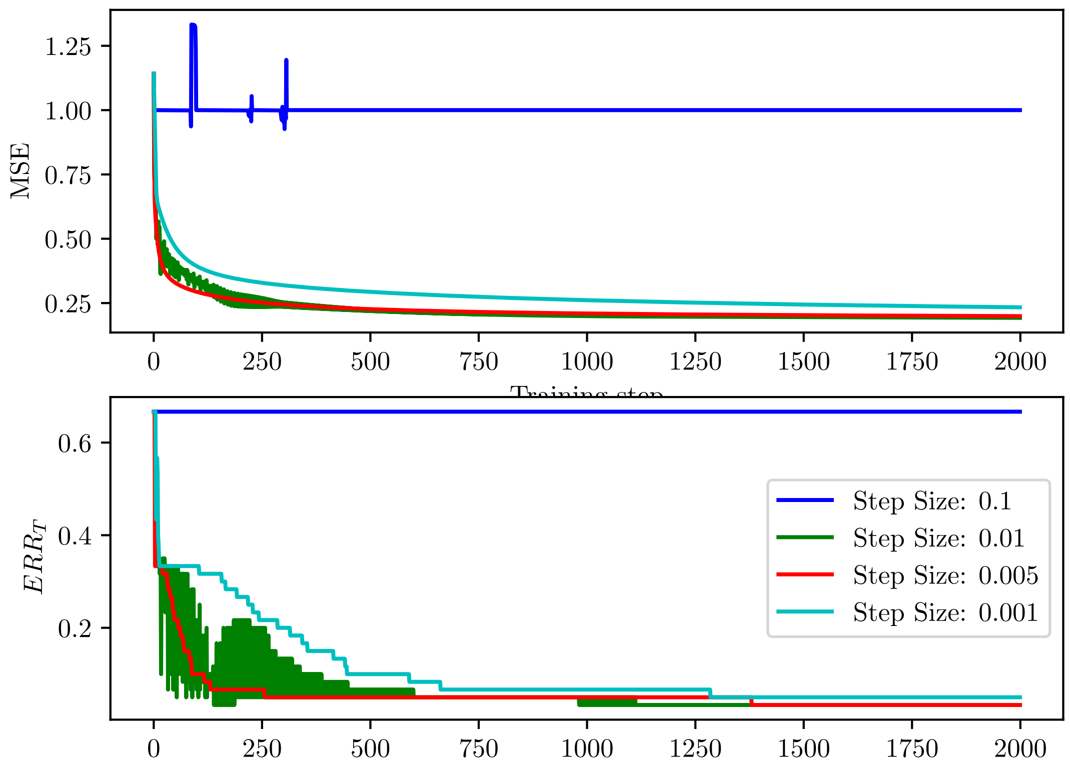


Figure 1: ERR_T and MSE evaluated at each training step for classifiers with different step sizes.

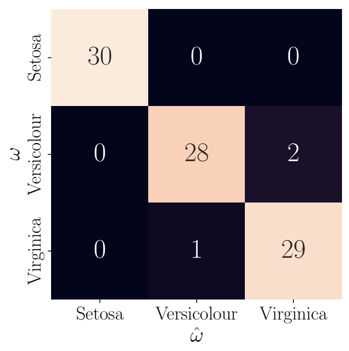
each class as training set and the last 20 as test set (later referred to as set partitioning I), and using the last 30 samples of each class as a training set and the first 20 as test set (later referred to as set partitioning II). The resulting error rates are shown in Table 1. In all further experiments, set partitioning I is used.

2.3 Results

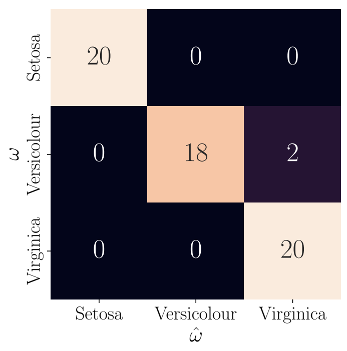
The confusion matrices that result from training the classifier on the different set partitionings can be seen in Figs. 2 and 3. These show that there are discrepancies between the two datasets, which is discussed in section 2.4.

Table 1: Error rates for the different set partitionings using a classifier trained for 1500 iterations with $\alpha = 0.005$.

set partitioning	ERR_D [%]	ERR_T [%]	95% CI of p_e [%]
I	3.33	3.33	[0.92, 11.36]
II	5.56	1.67	[0.29, 8.86]

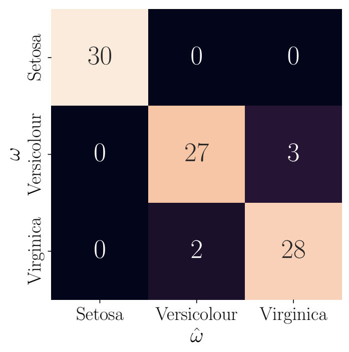


(a) Training set

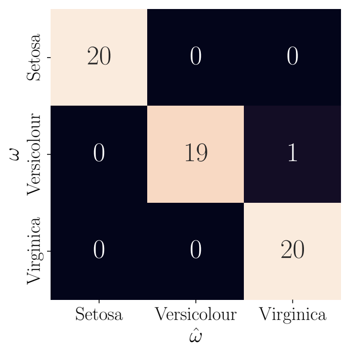


(b) Test set

Figure 2: Confusion matrices for set partitioning I. The number in each cell represents the number of samples belonging to ω that are classified as $\hat{\omega}$.



(a) Training set



(b) Test set

Figure 3: Confusion matrices for set partitioning II. The number in each cell represents the number of samples belonging to ω that are classified as $\hat{\omega}$.

The end results of the different error rates can be seen in Table 1. To explore the distribution of the features within the dataset, histograms of each feature were made for each class. These can be seen in Fig. 5. Then, to analyse the dataset's separability, correlation coefficients and scatter plots between different features are shown in Fig. 6.

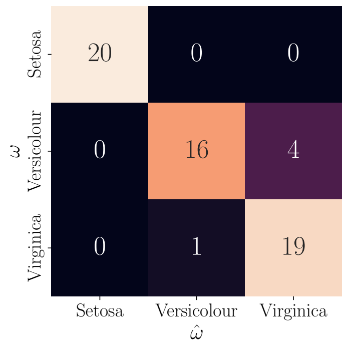
To investigate the effects of modifying the dataset, experiments with removing single features were conducted, giving the results in Table 2.

Table 2: Error rates when removing single features from the dataset.

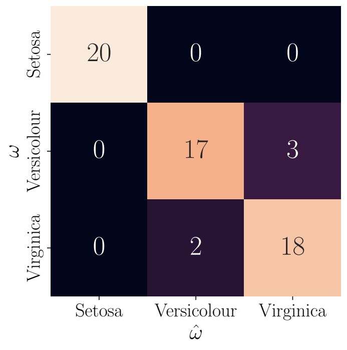
Removed feature	ERR_D [%]	ERR_T [%]	95% CI of p_e [%]
No features removed	3.33	5.00	[2.06, 11.62]
sepal length	4.44	3.33	[0.92, 11.36]
sepal width	3.33	3.33	[0.92, 11.36]
petal length	8.89	8.33	[3.61, 18.07]
petal width	4.44	8.33	[3.61, 18.07]

Table 3: Error rates when removing multiple features from the dataset.

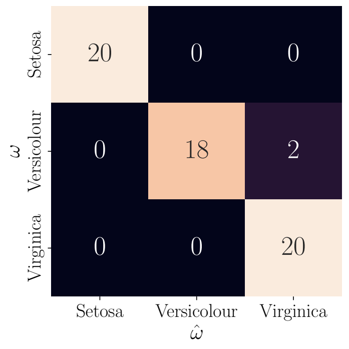
Removed features	ERR_D [%]	ERR_T [%]	95% CI of p_e [%]
Sepal width	3.33	3.33	[0.92,11.36]
Sepal length and sepal width	13.33	8.33	[3.61,18.07]
Sepal length, sepal width and petal length	12.22	8.33	[3.61,18.07]



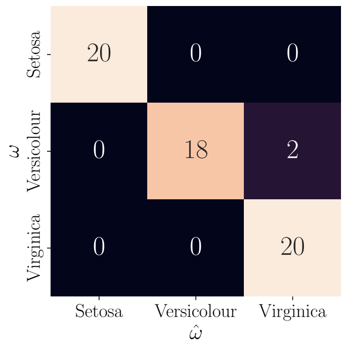
(a) Confusion matrix with only petal width.



(b) Confusion matrix with petal length and width.



(c) Confusion matrix with only sepal width removed.



(d) Confusion matrix with all features.

Figure 4: confusion matrices different features removed. The number in each cell represents the number of samples belonging to ω that are classified as $\hat{\omega}$.

2.4 Discussion

Table 1 and Figs. 2 and 3 show a difference in performance between the two set partitionings. Set partitioning I has the same performance when training as when testing, while set partitioning II shows better performance when testing than training as seen in 1. This may indicate that the test and training set from partitioning I have a more similar distribution than those from partitioning II. However, the performance difference is only indicated by two more misclassifications in training set II and one less misclassification in test set II, which, as the confidence intervals also suggest, makes it difficult to draw any strong conclusion.

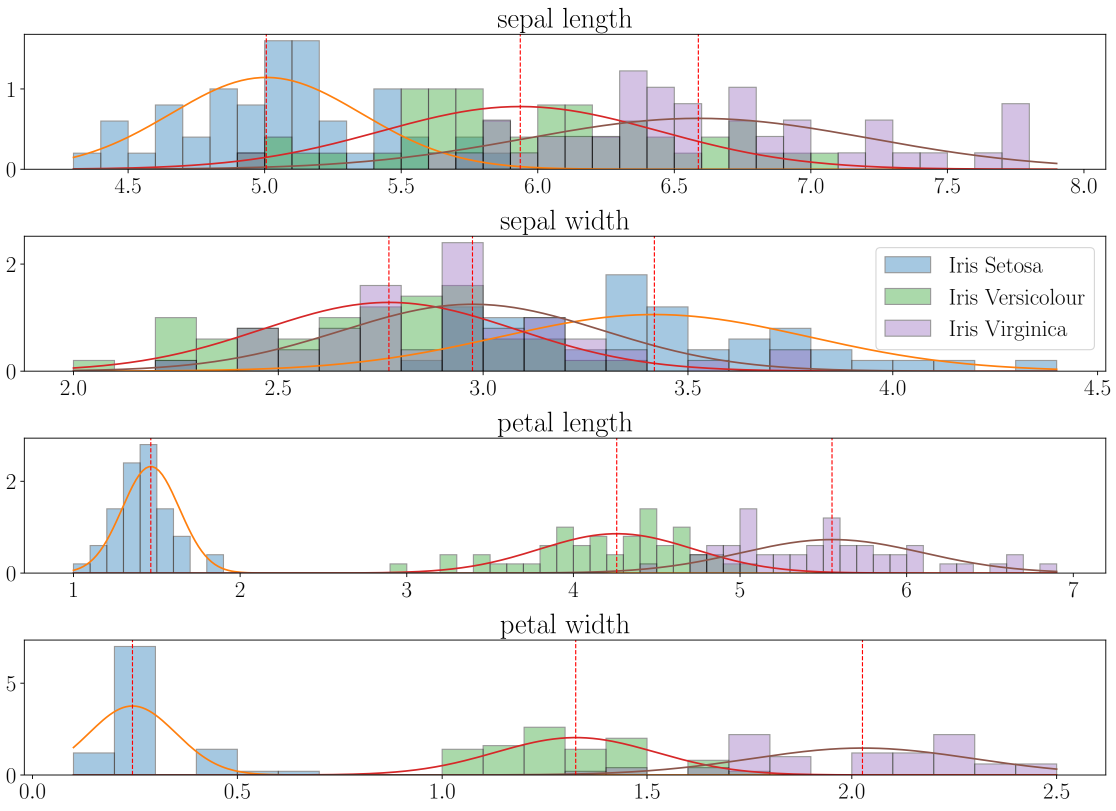


Figure 5: Histograms of each feature, colored after class. Gaussian estimates are overlaid for illustration purposes.

As mentioned in section 2.1, one may benefit from removing certain features from the dataset. Appropriate features to remove are often those whose histograms show the most overlap, and whose removal result in actual performance increase. From these criteria, and the results from Table 2 and Fig. 5, we conclude that sepal width is an appropriate first feature to remove. If more features are to be removed, the second one should be sepal length and third, petal length. The error rates for removing features in this order can be seen in Table 3. Removing only sepal width, we see a small improvement in ERR_T, suggesting that this feature contributes with little information for classification. Removing more features, we start seeing a more notable increase in error rate. This is expected, as feature removal at some point must lead to information loss.

Looking at Fig. 4, we see that *Iris Setosa* is consistently classified correctly, confirming its linear separability. Instead, all misclassifications occur between the *Iris Versicolour* and *Iris Virginica*. This issue is further illustrated in the histogram of Fig. 5 and the scatter plots in Fig. 6. These plots corroborate that, unlike *Iris Setosa*, *Iris Versicolour* and *Iris Virginica* are not perfectly linearly separable.

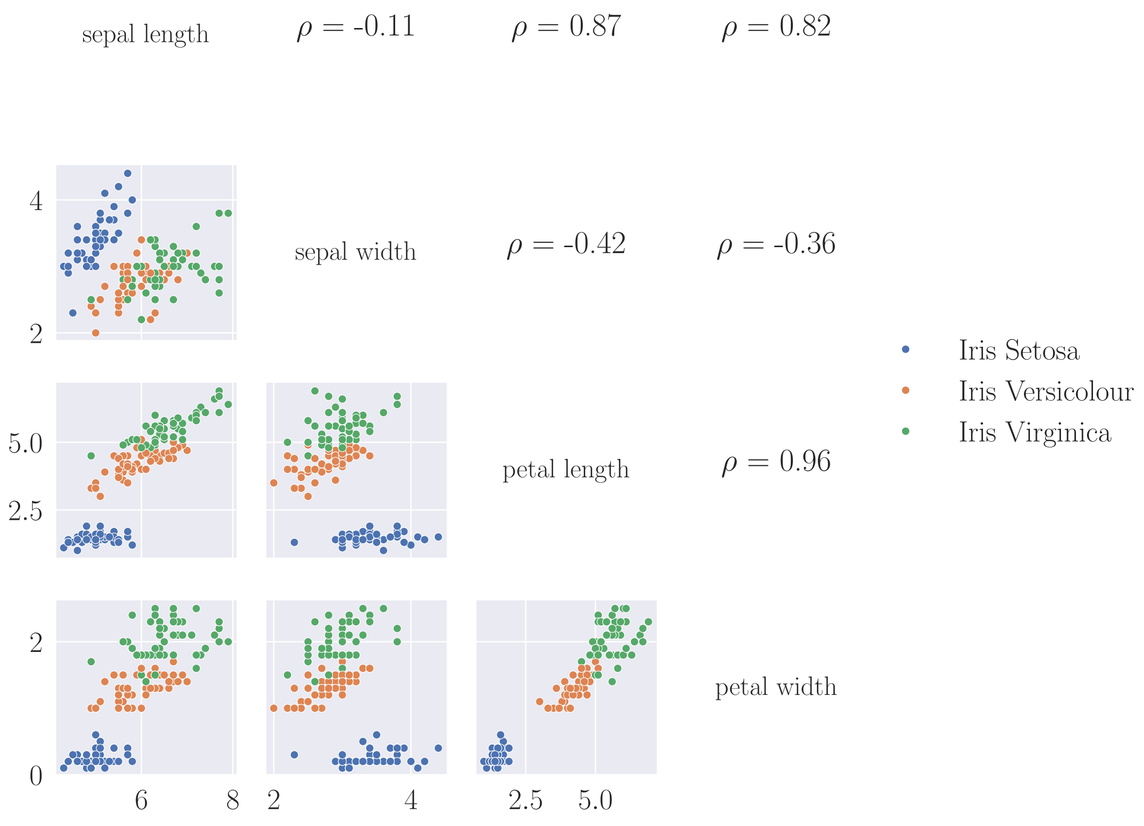


Figure 6: Pair plot with scatter plots of the features under the diagonal and the Pearson correlation coefficient ρ above the diagonal.

3 Classification of the MNIST Dataset

3.1 Theory

The MNIST dataset consists of 70000 labeled images of handwritten digits, partitioned into 60000 training samples and 10000 test samples. Each image is 28×28 pixels, giving 784 features and 10 classes.

In our experiments, three different classification methods were applied to this dataset: k-Nearest Neighbours (k-NN) with all training samples as templates and $k = 1$, and $k-$NN with K-means clustered training samples as templates with $k = 1$ and $k = 7$.

$k-$NN is a template based, non-parametric classification method for supervised learning, first proposed by Fix and Hodges in 1951 [1].

The simplest case of k-NN is when $k = 1$, sometimes abbreviated Nearest Neighbour (NN); When classifying an input $\boldsymbol{x}$ into one of M classes $\omega_1, ..., \omega_M$, given N templates $\left\{\left\{\underline{\boldsymbol{x}}_1, \omega_{c(1)}\right\}, ..., \left\{\underline{\boldsymbol{x}}_N, \omega_{c(N)}\right\}\right\}$, we choose the class of the nearest template to $\boldsymbol{x}$ due to some distance measure $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Mathematically, this can be stated as

$$\hat{\omega} = \omega_{c(n^*)}, \text{ where } n^* = \arg\min_n d(\boldsymbol{x}, \underline{\boldsymbol{x}}_n). \tag{14}$$

In the general case, when k can be any integer above 1, we instead choose the templates associated with the k lowest distances, and choose $\hat{\omega}$ as the one that the majority of these k neighbouring templates belong to. In the case that there is a similar number of neighbouring templates belonging to multiple classes, the class of these belonging to the nearest neighbour of $\boldsymbol{x}$ is chosen [2].

An essential detail to k-NN is the definition of the distance measure d. In our experiments we use the squared Euclidean distance,

$$d(\boldsymbol{x}_1, \boldsymbol{x}_2) = (\boldsymbol{x}_1 - \boldsymbol{x}_2)^T (\boldsymbol{x}_1 - \boldsymbol{x}_2). \tag{15}$$

When selecting a value of k, there is an important trade-off to consider: In the general case, a low value of k creates a classifier more prone to overfitting, meaning that it may fail to generalize the feature space between templates. A classifier with higher k often generalizes better, but requires more templates and leads to more computational complexity. In practice, however, the optimal k is very dependent on the dataset, and is often best chosen by trial and error [5].

K-means clustering is a form of *unsupervised* learning, meaning that it aims to find patterns in unlabeled data. More specifically, it finds a set of K cluster centers $\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_K$ that minimize the cost function

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{n,k} (\boldsymbol{x}_n - \boldsymbol{\mu}_k)^T (\boldsymbol{x}_n - \boldsymbol{\mu}_k), \tag{16}$$

where $r_{n,k}$ is the *indicator function*

$$r_{n,k} = \begin{cases} 1, & \boldsymbol{x}_n \text{ belongs to } \boldsymbol{\mu}_k \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

The optimal clustering can then be interpreted as the set of centers which minimize the sum of distances from each sample $\boldsymbol{x}_n$ to its nearest cluster center. Equation (16) is usually minimized by the Expectation Maximization (EM) algorithm outlined in 1.

Algorithm 1 EM algorthm for K-means clustering.

```
Choose $K$ initial centers $\{\boldsymbol{\mu}_1(0), ..., \boldsymbol{\mu}_K(0)\}$ randomly.
while $\{\boldsymbol{\mu}_1(t), ..., \boldsymbol{\mu}_K(t)\}$ have not converged do
  for $n = 1, ..., N$ do
    Define $\boldsymbol{x}_n$ to belong to the closest center $\boldsymbol{\mu}_k(t)$ and set $r_n$ accordingly.
  end for
  for $k = 1, ..., K$ do
    Set $\boldsymbol{\mu}_k(t+1)$ equal to the barycenter of all $\boldsymbol{x}_n$ belonging to $\boldsymbol{\mu}_k(t)$.
  end for
end while
```

Classification of handwritten digits is, however, a problem of supervised learning. Therefore, K-means clustering is only used as a method of preprocessing the training data into a smaller set of templates to be used in k-NN. This is done by performing K-means within each class, and using the resulting cluster centers as templates for the corresponding class. The expected outcome of this is a classifier with inferior accuracy, as the transition from N to $KM \ll N$ templates involves an inevitable information loss. However if the clustering is successful, this decrease in accuracy should be small compared to the computational performance gain of having fewer templates [2]. This new template space is now significantly less densely populated, so it is expected that this method requires a smaller k than a pure k-NN classifier.

The choice of K in K-means is highly dependent on the dataset [6]. A higher K gives less information loss and therefore a more accurate representation of the original data, but uses more time in the clustering process, as well as in subsequent classification. One method for determining a suitable value is by the so-called elbow method [4]. Here, clustering the data is done for a range of values of K and the final value of the cost function J^* is compared. A lower value of J^* translates to less information loss, so such a graph can be helpful to arrive at a suitable trade-off between accuracy and speed, which is often found in the “elbow” of the J-curve.

3.2 Implementation

All classification methods were implemented in python, and code is given in Appendix C.

Well optimized implementations of both k-NN and K-means are provided in the module `scikit-learn`, through the classes `KNeighborsClassifier` and `KMeans`. These provide pre-compiled code and specialized data structures that significantly reduce execution time.

Clustering of templates was performed using `KMeans`. However, for the learning opportunity and for our personal enjoyment, k-NN was implemented "from scratch" using `numpy` arrays, which provide a well optimized framework for matrix operations. Two different implementations were made, one for running k-NN with cluster centers as templates, and one for running NN with raw samples as templates. The latter runs predictions in chunks of 1000 samples to achieve a reasonable memory usage.

The implementation using raw samples as templates only provides functionality for NN and not k-NN. This is because an efficient implementation of k-NN for many templates is more complex, and only NN was requested in our task description [3]. We were, however, still curious to examine the effects of letting $k > 1$, so this was done by evaluating the performance of a k-NN classifier for $k = 1, ..., 20$, using `scikit-learn`'s `KNeighborsClassifier`.

Similarly, we were instructed to perform K-means with $K = 64$ [3]. Again, we were curious of the effects of changing this value, so clustering and subsequent classification with NN was performed with $K = 1, ..., 128$.

All other experiments were executed with $K = 64$.

3.3 Results

Varying k in k-NN with and without clustering gave the results in Fig. 7. Varying K in K-means gave the results in Fig. 8.

Among the three classification methods, the lowest error rate was achieved using NN without clustering, where $ERR_T = 3.9\%$. Clustering the templates gave a lower accuracy, but with a significant performance gain. The error rates and computation time for classifying all samples in the test set (t_T) is given for all three methods in Table 4.

Some examples of correctly and incorrectly classified digits using NN with and without clustering are shown in Figs. 9 and 10, respectively. The test set confusion matrices from all three methods are shown in Figs. 11, 12 and 13.

3.4 Discussion

Table 4 shows that all three attempted methods are viable classifiers for the MNIST dataset. There are nonetheless important differences between the methods: NN without clustering shows a superior accuracy out of the three,

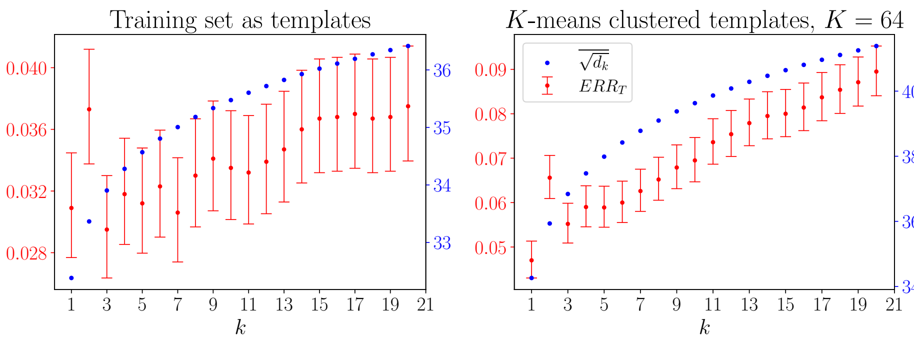


Figure 7: Mean Euclidian distance to kth nearest neighbour $\overline{\sqrt{d_k}}$ over all test samples, along with test set error rates, using a k-NN classifier with and without clustering. Error bars show estimated 95% confidence interval of p_e.

Table 4: Error rates and total test set computation time t_T for each classification method.

Classification method	ERR_T [%]	95% CI of p_e [%]	t_T [m:s]
NN, no clustering	3.09	[2.77, 3.45]	25:43.4
NN, clustered templates	4.70	[4.30, 5.13]	00:37.6
7-NN, clustered templates	5.47	[5.04, 5.93]	00:36.6

and the difference is statistically significant with $p = 0.05$. But as expected, this accuracy comes at the cost of classification speed; Using our own implementation of k-NN and `scikit-learn`'s `KMeans`, NN without clustering was more than 41 times slower than with clustering, when classifying the test set – including the time spent on clustering the training set. 7-NN with clustering had approximately the same speed increase, but a significantly higher error rate than NN with clustering. This effectively rules out 7-NN with clustering as a preferable classification method for this dataset.

Fig. 7 further shows that this result is part of a trend, where error rate increases with larger values of k – both with and without clustered templates. One likely cause of this is that both classifiers have too few templates to populate the feature space densely enough. This is also illustrated by the steep increase in distances to kth nearest neighbour; When going from $k = 1$ to $k = 3$, templates that are notably further from $\boldsymbol{x}$ in the feature space will affect $\hat{\omega}$. The effect of template density also explains why the increase in both ERR_T and $\overline{\sqrt{d_k}}$ is more prevalent in when using clustered templates, in which case the number of templates is reduced from 60000 to $10 \cdot K = 640$.

Examining the effects of K in K-means, we consider Fig. 8. As expected, when increasing K, J^* decreases and clustering time increases. Also as

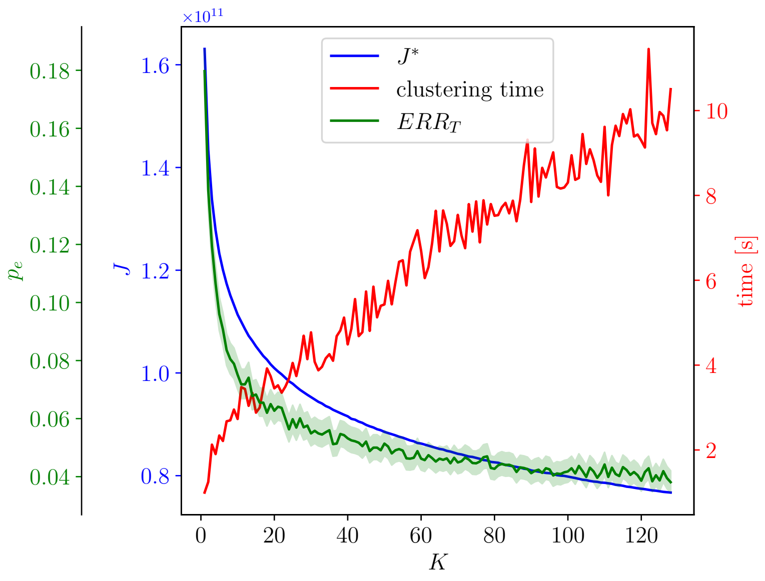


Figure 8: Final cost function value J^* and total time needed to cluster the entire training set using K-means, along with test set error using cluster centers as templates. The shaded region shows an estimated 95% confidence interval of p_e for each K.

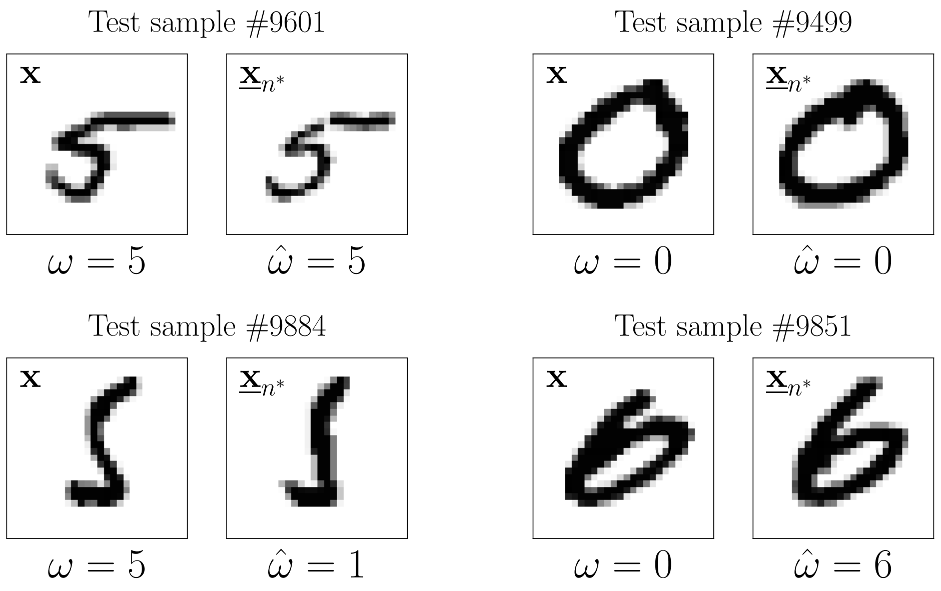


Figure 9: Some examples of test samples $\boldsymbol{x}$ and their nearest neighbours $\underline{\boldsymbol{x}}_{n^*}$. Here, $\hat{\omega}$ is found using NN without clustering.

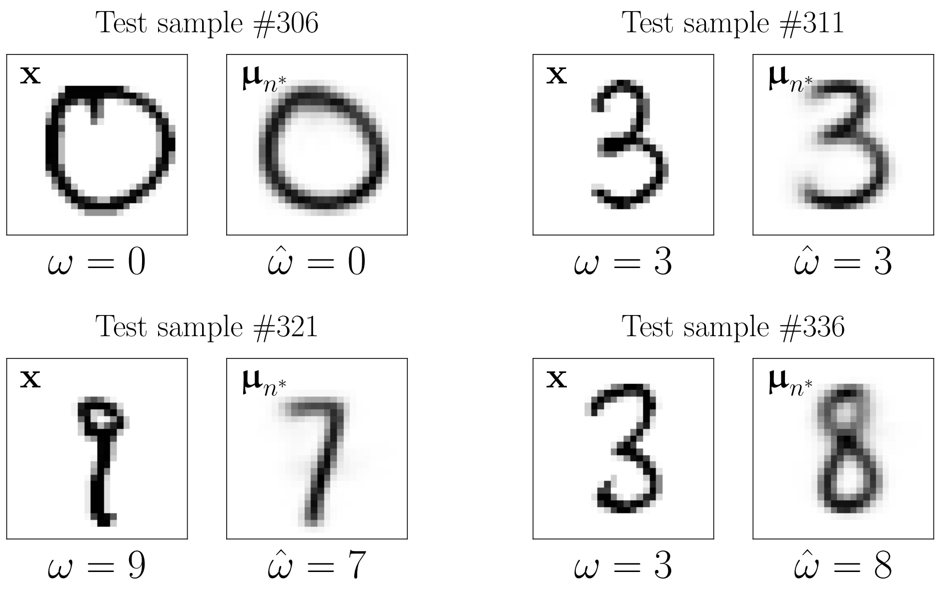


Figure 10: Some examples of test samples $\boldsymbol{x}$ and their nearest cluster centers $\boldsymbol{\mu}_{n^*}$. Here, $\hat{\omega}$ is found using NN with clustering.

expected, ERR_T follows the decrease in J^*. From this plot we can see that $K = 64$ marks an appropriate trade-off between speed and accuracy; Both J^* and ERR_T start to flatten out, while clustering time continues to increase seemingly linearly. The confidence interval further indicates that the decreases in ERR_T become less statistically significant beyond $K = 64$.

To gain some intuitive understanding of the weaknesses of the classifiers, we review the confusion matrices seen in Figs. 11-13. Here, we can see that the digit that is most often misclassified using NN without clustering is 8, with $ERR_T \approx 0.945$. Interestingly, when clustering the templates, we see a notable accuracy decrease for almost all digits *except* 8. This suggests that the class 8 has templates that cluster well; That is, the clusters represent the training set with little information loss. This is in contrast to digits like 3, that are notably more often misclassified when templates are clustered.

Furthermore, the confusion matrices reveal that some digits are more often confused with each other. For example, regardless of classification method, the digits 7 and 4 are often confused with 9, and opposite. In contrast, 5 and 2 are practically never confused with each other. This makes intuitive sense, as Euclidean distance is a measure related to graphical similarity, and 7 and 4 both are graphically quite similar to 9, while 5 and 2 are almost mirror images of each other. This concept of graphical similarity is demonstrated in Figs. 9 and 10, where we can clearly see that the classifier finds templates that closely resemble the digit to be classified. The misclassifications displayed in Fig. 9 can be explained as being due to outliers in the dataset; For sample #9884 the nearest template could be called an outlier, as it would be difficult even for a human to tell whether it is a 1 or a 5. For sample #9851, the sample itself could be called an outlier, as it "looks" more like a 6 than a 0. The misclassifications in Fig. 9 are instead due to a weakness in using Euclidean distance as a similarity measure; The measure has no concept of a circle, or closed loop, so for sample #321, a 9 is classified as a 7 partly because the circle of the digit 9 has a similar footprint to the bar of the digit 7 in the template. For sample #336, the digit 3 covers most of the same pixels as the digit 8 in the template, and is therefore misclassified. In both of these cases, most humans could easily correctly classify the digits.

To counteract some of these weaknesses of the Euclidean distance, one could for instance attempt to use the Mahalanobis distance instead, which takes the covariances of the features into account. However, this requires estimation of said covariances, which is no trivial task in a dataset of limited size and with a relatively large feature space such as this one [2].

4 Conclusion

For classification in the Fisher's Iris dataset, we employed a linear classifier optimized using gradient descent with MSE as objective function. The MSE gradient was computed using a batch size equal to the length of the training set, with an empirically determined step size α of 0.005, and for 1500 training steps. The classifier was implemented in python code, and trained and evaluated on two different partitionings of the dataset, giving classification errors ERR_T of 1.67% and 3.33%.

We found that each feature contributes with a different amount of information for classification, and that especially the feature sepal width had little impact on classification accuracy. However, when removing more than one feature, accuracy deteriorated notably. Although notable, the different values of ERR_T fall well within each other's estimated confidence intervals. This points to a repeating issue with the Iris dataset; With its modest 150 samples, definitive conclusions are hard to reach when comparing different methods.

Despite these limitations, the task demonstrated that the Iris dataset is almost linearly separable. More specifically, in this specific dataset the *Iris Setosa* class is perfectly linearly separable from the other two classes, and *Iris Virginica* and *Iris Versicolor* nearly so.

In the MNIST classification problem, we utilized mainly three variants of a k-NN classifier with Euclidean distance; NN with all training samples as templates, and NN and 7-NN with K-means clustered training samples as templates. All of these were implemented in python, using the library `scikit-learn`'s function `KMeans` for clustering.

For K-means clustering, a value of $K = 64$ was recommended in our task description, and when further investigated, this value seemed appropriate. Clustering was applied within each of the ten classes, giving 640 templates. Among all three methods, NN without clustering gave the lowest error rate with $ERR_T = 3.09\%$, NN with clustering gave $ERR_T = 4.70\%$, and 7-NN with clustering gave $ERR_T = 5.47\%$. Both methods using clustered templates came with a processing time reduction of more than 97% compared to NN without clustering. This makes NN with clustering a viable method in situations where speed is a requirement. The same can not be said for 7-NN, which with approximately the same speed increase gave worse accuracy.

For completeness, several values of k in k-NN were tested, both with and without clustering. From these experiments we concluded that $k = 1$ was among the most suitable values in both cases.

To conclude; We were able to construct viable classifiers for both tasks. For the Iris dataset, we reached high accuracy with several different variations of the dataset. Although the small dataset size makes it difficult to compare results, we were able to identify that some features have a notably lower effect on classification accuracy than others, and somewhat acceptable results were even reached removing all features except one. For the MNIST classification problem, the challenge of balancing accuracy and speed became apparent. Although this report only investigates a few classical and rather simple classification methods, this trade-off is relevant for most classification tasks with large datasets.

References

[1] Evelyn Fix and J. L. Hodges. “Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties.” In: *International Statistical Review / Revue Internationale de Statistique* 57.3 (1989), pp. 238–247. URL: `http://www.jstor.org/stable/1403797` (visited on 04/28/2024).

[2] Magne H. Johnsen. *Classification.* NTNU, Dec. 2017. (Visited on 04/20/2024).

[3] Magne H. Johnsen. *Project descriptions and tasks in classification.* NTNU, Feb. 2018. (Visited on 04/21/2024).

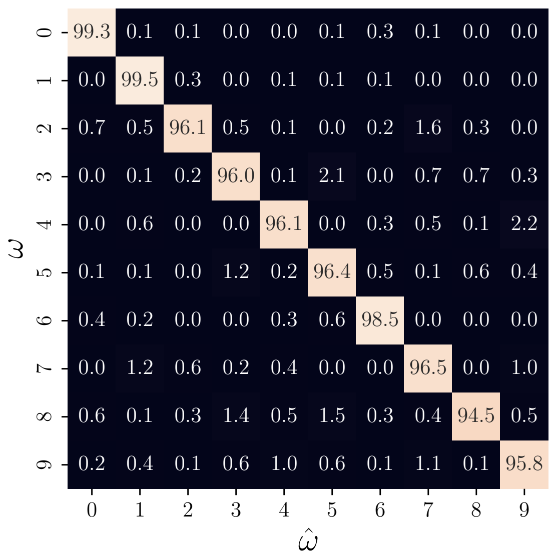
[4] Trupti Kodinariya and Prashant Makwana. “Review on Determining of Cluster in K-means Clustering.” In: *International Journal of Advance Research in Computer Science and Management Studies* 1 (Jan. 2013), pp. 90–95. (Visited on 04/24/2024).

[5] Iman Paryudi. “What Affects K Value Selection In K-Nearest Neighbor.” In: *International Journal of Scientific & Technology Research* 8 (2019), pp. 86–92. URL: `https://api.semanticscholar.org/CorpusID:203992542` (visited on 04/29/2024).

[6] Pierluigi S. Rossi. *Estimation, Detection and Classification. Lecture 14-15: Classification and Classification Systems.* NTNU, May 2024. (Visited on 04/28/2024).

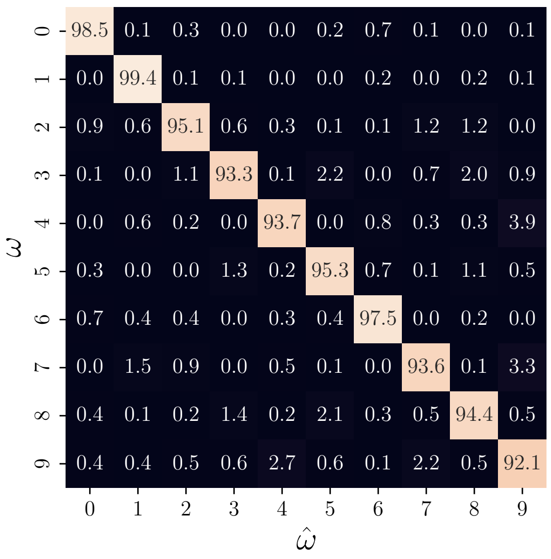
[7] Edwin B. Wilson. “Probable Inference, the Law of Succession, and Statistical Inference.” In: *Journal of the American Statistical Association* 22.158 (1927), pp. 209–212. URL: `http://www.jstor.org/stable/2276774` (visited on 04/28/2024).

A MNIST confusion matrices



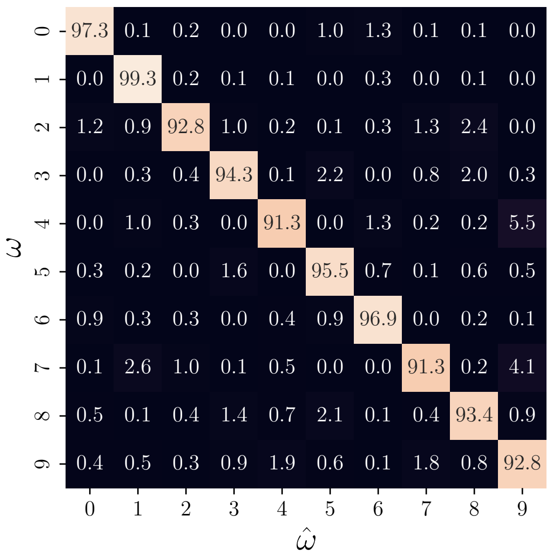
ω \ $\hat{\omega}$	0	1	2	3	4	5	6	7	8	9
0	99.3	0.1	0.1	0.0	0.0	0.1	0.3	0.1	0.0	0.0
1	0.0	99.5	0.3	0.0	0.1	0.1	0.1	0.0	0.0	0.0
2	0.7	0.5	96.1	0.5	0.1	0.0	0.2	1.6	0.3	0.0
3	0.0	0.1	0.2	96.0	0.1	2.1	0.0	0.7	0.7	0.3
4	0.0	0.6	0.0	0.0	96.1	0.0	0.3	0.5	0.1	2.2
5	0.1	0.1	0.0	1.2	0.2	96.4	0.5	0.1	0.6	0.4
6	0.4	0.2	0.0	0.0	0.3	0.6	98.5	0.0	0.0	0.0
7	0.0	1.2	0.6	0.2	0.4	0.0	0.0	96.5	0.0	1.0
8	0.6	0.1	0.3	1.4	0.5	1.5	0.3	0.4	94.5	0.5
9	0.2	0.4	0.1	0.6	1.0	0.6	0.1	1.1	0.1	95.8

Figure 11: Confusion matrix from classifying the test set with NN without clustering. The number in each cell represents the percentage of samples belonging to ω that are classified as $\hat{\omega}$.



$\omega \backslash \hat{\omega}$	0	1	2	3	4	5	6	7	8	9
0	98.5	0.1	0.3	0.0	0.0	0.2	0.7	0.1	0.0	0.1
1	0.0	99.4	0.1	0.1	0.0	0.0	0.2	0.0	0.2	0.1
2	0.9	0.6	95.1	0.6	0.3	0.1	0.1	1.2	1.2	0.0
3	0.1	0.0	1.1	93.3	0.1	2.2	0.0	0.7	2.0	0.9
4	0.0	0.6	0.2	0.0	93.7	0.0	0.8	0.3	0.3	3.9
5	0.3	0.0	0.0	1.3	0.2	95.3	0.7	0.1	1.1	0.5
6	0.7	0.4	0.4	0.0	0.3	0.4	97.5	0.0	0.2	0.0
7	0.0	1.5	0.9	0.0	0.5	0.1	0.0	93.6	0.1	3.3
8	0.4	0.1	0.2	1.4	0.2	2.1	0.3	0.5	94.4	0.5
9	0.4	0.4	0.5	0.6	2.7	0.6	0.1	2.2	0.5	92.1

Figure 12: Confusion matrix from classifying the test set with NN with clustering. The number in each cell represents the percentage of samples belonging to ω that are classified as $\hat{\omega}$.



$\omega \backslash \hat{\omega}$	0	1	2	3	4	5	6	7	8	9
0	97.3	0.1	0.2	0.0	0.0	1.0	1.3	0.1	0.1	0.0
1	0.0	99.3	0.2	0.1	0.1	0.0	0.3	0.0	0.1	0.0
2	1.2	0.9	92.8	1.0	0.2	0.1	0.3	1.3	2.4	0.0
3	0.0	0.3	0.4	94.3	0.1	2.2	0.0	0.8	2.0	0.3
4	0.0	1.0	0.3	0.0	91.3	0.0	1.3	0.2	0.2	5.5
5	0.3	0.2	0.0	1.6	0.0	95.5	0.7	0.1	0.6	0.5
6	0.9	0.3	0.3	0.0	0.4	0.9	96.9	0.0	0.2	0.1
7	0.1	2.6	1.0	0.1	0.5	0.0	0.0	91.3	0.2	4.1
8	0.5	0.1	0.4	1.4	0.7	2.1	0.1	0.4	93.4	0.9
9	0.4	0.5	0.3	0.9	1.9	0.6	0.1	1.8	0.8	92.8

Figure 13: Confusion matrix from classifying the test set with 7-NN with clustering. The number in each cell represents the percentage of samples belonging to ω that are classified as $\hat{\omega}$.

B Iris classification code

```
------------------------------------------------------------
                    classifier.py
------------------------------------------------------------
import numpy as np


class classifier:
    def __init__(self, C, D):
        mu = 0.0
        sigma = 0.1
        self.W = np.random.normal(mu, sigma, (C, D +
            ↪ 1))
        self.C = C

    def gradient(self, x, g, t):
        return np.outer((g - t) * g * (1 - g), x.T)

    def evaluate(self, x):
        return sigmoid(self.W @ x)

    def train(self, x, t, step):
        grad = np.zeros_like(self.W)
        errors = 0
        for xk_, tk in zip(x, t):
            xk = np.append(xk_, 1)
            g = self.evaluate(xk)
            grad += self.gradient(xk, g, tk)
            if np.argmax(g) != np.argmax(tk):
                errors += 1
        self.W = self.W - step * grad
        return errors / len(x)

    def confusion(self, x, t):
        conf = np.zeros((self.C, self.C))
        for xk_, tk in zip(x, t):
            xk = np.append(xk_, 1)
            g = self.evaluate(xk)
            truth = np.argmax(tk)
            guess = np.argmax(g)
            conf[truth, guess] += 1
        return conf

    def validate(self, x, t):
        errors = 0
        for xk_, tk in zip(x, t):
            xk = np.append(xk_, 1)
            g = self.evaluate(xk)
            truth = np.argmax(tk)
            guess = np.argmax(g)
            if guess != truth:
                errors += 1
        return errors / len(x)

    def train_on_dataset(self, train_x, train_y, N,
       ↪ step_size, test_x, test_y):
        train_err = np.zeros(N)
        test_err = np.zeros(N)
        assert len(train_x) == len(train_y)
        for i in range(N):
            p = np.random.permutation(len(train_x))
            err = self.train(train_x[p, :], train_y[p
               ↪ , :], step_size)
            train_err[i] = err
            test_err[i] = self.validate(test_x,
               ↪ test_y)
            print(f"TRAINING... {i}/{N}: \t{100*err
               ↪ :.2f}", end="\r", flush=True)

        print("TRAINING... DONE
           ↪ ")
        return train_err, test_err


def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
```

```
-----------------------------------------------------
                    data.py
-----------------------------------------------------
import pandas as pd
import numpy as np

class_names = ["Iris Setosa","Iris Versicolour", "
    ↪ Iris Virginica"]
feature_names = ["sepal length","sepal width","petal
    ↪ length","petal width"]
class_names_short = ["Setosa","Versicolour", "
    ↪ Virginica"]
file_paths = ["class_1", "class_2", "class_3"]
n_features = 4
n_classes = 3

def onehot_encode(i, N, rep=1):
    if rep > 1:
        y = np.zeros((rep, N))
        y[:, i] = 1
        return y
    else:
        y = np.zeros(N)
        y[i] = 1
        return y

def drop_feature(data, feature):
    assert (feature in feature_names)
    feature_index = feature_names.index(feature)
    return np.delete(data, feature_index, axis=1)

def new_names(features):
    return [n for n in feature_names if n not in
        ↪ features]

def load_data(var=1, drop_features=()):
    train_x = []
    test_x = []
    train_y = []
    test_y = []

    if isinstance(drop_features, str):
        drop_features = (drop_features, )

```

```python
    for i, file_path in enumerate(file_paths):
        data = pd.read_csv(file_path, header=None).to_numpy()
        for f in drop_features:
            data = drop_feature(data, f)

        if var == 1:
            train = data[:30]
            test = data[30:]
        elif var == 2:
            train = data[20:]
            test = data[:20]
        else:
            raise Exception

        train_x.append(train)
        test_x.append(test)
        train_y.append(onehot_encode(i, n_classes, rep=30))
        test_y.append(onehot_encode(i, n_classes, rep=20))

    train_x = np.vstack(train_x)
    train_y = np.vstack(train_y)
    test_x = np.vstack(test_x)
    test_y = np.vstack(test_y)
    x = np.vstack((train_x, test_x))
    y = np.vstack((train_y, test_y))

    return train_x, train_y, test_x, test_y, x, y
```

```
---------------------------------------------------
                   main.py
---------------------------------------------------
from classifier import classifier
from data import *

np.random.seed(0)
remove_features = []
train_x, train_y, test_x, test_y, x, y = load_data(1,
    ↪  remove_features)
c = classifier(n_classes, n_features - len(
    ↪ remove_features))
c.train_on_dataset(train_x, train_y, 1500, 0.005,
    ↪ test_x, test_y)
```

C MNIST classification code

```
---------------------------------------------------
                 classifier.py
---------------------------------------------------
import numpy as np
import sklearn.neighbors
from tqdm import tqdm, trange
import sklearn.cluster
import collections


class NNClassifier:
    def __init__(self, template_x, template_y,
        ↪ chunk_size=None):
        self.n_templates = len(template_x)

        self.template_x = template_x
        self.template_y = template_y

        if chunk_size is None:
            self.chunk_size = self.n_templates
        else:
            self.chunk_size = chunk_size
        self.n_chunks = int(np.ceil(self.n_templates
            ↪ / self.chunk_size))
        self.template_x_chunked = list(
            [
```

```python
                self.template_x[
                    i
                    * self.chunk_size : min(
                        self.n_templates, ((i + 1) * self.chunk_size)
                    )
                ]
                for i in range(self.n_chunks)
            ]
        )
        self.template_y_chunked = list(
            [
                self.template_y[
                    i
                    * self.chunk_size : min(
                        self.n_templates, ((i + 1) * self.chunk_size)
                    )
                ]
                for i in range(self.n_chunks)
            ]
        )

    def get_nn(self, x):
        dist = np.sum(np.square(x - self.template_x), axis=1)
        idx = np.argmin(dist)
        return self.template_y[idx], self.template_x[idx]

    def predict_single(self, x):
        dist = np.sum(np.square(x - self.template_x), axis=1)
        idx = np.argmin(dist)
        return self.template_y[idx]

    def predict_chunk(self, x):
        minima = np.zeros((len(x), self.n_chunks))
        minima_labels = np.zeros_like(minima, dtype=int)
        for i in trange(self.n_chunks):
            diff = x[:, np.newaxis, :] - self.template_x_chunked[i][np.newaxis, :, :]
            dist = np.sum(np.square(diff), axis=2)
            min_idx = np.argmin(dist, axis=1)
            minima[:, i] = dist[range(len(x)), min_idx]
            minima_labels[:, i] = self.template_y_chunked[i][min_idx, 0]
        glb_min_idx = np.argmin(minima, axis=1)
        predictions = minima_labels[range(len(x)), glb_min_idx]
        return predictions

    def predict_array(self, x, chunk_size=None):
        N = len(x)
        if chunk_size is None:
            chunk_size = N
        n_chunks = int(np.ceil(N / chunk_size))
        chunked_x = list(
            [
                x[i * chunk_size : min(N, ((i + 1) * chunk_size))]
                for i in range(n_chunks)
            ]
        )
        predictions = np.hstack(
            [self.predict_chunk(chunk) for chunk in tqdm(chunked_x)]
        )
        return predictions


class ClusteredKNNClassifier:
    def __init__(self, template_x, template_y, n_classes, clusters_per_class):
        self.n_classes = n_classes
        self.n_templates = len(template_x)

        self.template_x, self.template_y = ClusteredKNNClassifier.cluster_templates(
            template_x,
            template_y,
            clusters_per_class,
            n_classes,
        )
```

```python

    def cluster_templates(x, y, clusters_per_class, n_classes):
        cluster_centers = np.zeros((n_classes * clusters_per_class, x.shape[1]))
        cluster_labels = np.zeros((n_classes * clusters_per_class,), dtype=int)
        for i in range(n_classes):
            indices = np.where(y == i)[0]
            kmeans = sklearn.cluster.KMeans(n_clusters=clusters_per_class)
            kmeans.fit(x[indices])
            cluster_centers[i * clusters_per_class : (i + 1) * clusters_per_class] = (
                kmeans.cluster_centers_
            )
            cluster_labels[i * clusters_per_class : (i + 1) * clusters_per_class] = i
        return cluster_centers, cluster_labels

    def predict_single(self, x, K):
        dist = np.sum(np.square(x - self.template_x), axis=1)
        lowest_indices = np.argpartition(dist, K)[:K]
        lowest_dist = dist[lowest_indices]
        sort_indices = np.argsort(lowest_dist)
        lowest_dist = lowest_dist[sort_indices]
        lowest_labels = self.template_y[lowest_indices][sort_indices]
        pred = ordered_majority_vote(lowest_labels)
        return pred

    def predict_array(self, x, K):
        predictions = [self.predict_single(xk, K) for xk in tqdm(x)]
        return np.hstack(predictions)

    def get_nn(self, x):
        dist = np.sum(np.square(x - self.template_x), axis=1)
        idx = np.argmin(dist)
        return self.template_y[idx], self.template_x[idx]

```

```

def ordered_majority_vote(x):
    c = collections.OrderedDict()
    for i in x:
        if i in c:
            c[i] += 1
        else:
            c[i] = 1
    maxidx = np.argmax(list(c.values()))
    return list(c.keys())[maxidx]
```

```
----------------------------------------------------
                   main.py
----------------------------------------------------
import numpy as np
from classifier import NNClassifier,
    ↪ ClusteredKNNClassifier
import scipy
import time

def load_data():
    f = scipy.io.loadmat("data_all.mat")
    testv = np.array(f["testv"], dtype=np.int32)
    trainv = np.array(f["trainv"], dtype=np.int32)
    testlab = np.array(f["testlab"], dtype=np.int32)
    trainlab = np.array(f["trainlab"], dtype=np.int32
        ↪ )
    return trainv, trainlab, testv, testlab


def run_KNN_timed(c: ClusteredKNNClassifier, K):
    start = time.time()
    test_predictions = c.predict_array(test_x, K)
    test_time = time.time() - start
    return test_time


def run_NN_timed(c: NNClassifier):
    start = time.time()
    test_predictions = c.predict_array(test_x,
        ↪ chunk_size=1000)
    test_time = time.time() - start
    return test_time

train_x, train_y, test_x, test_y = load_data()

c = NNClassifier(
    train_x,
    train_y,
    chunk_size=1000,
)
run_NN_timed(c)

c = ClusteredKNNClassifier(
    train_x,
    train_y,
    n_classes=10,
    clusters_per_class=64,
)
run_KNN_timed(c, 1)
run_KNN_timed(c, 7)
```