



LUT University
Lappeenranta, Finland
December 5, 2025

BM40A0702 – Pattern Recognition and Machine Learning

Digits-3D

Feature Extraction and Classification System
for Hand-Drawn 3D Digits

Noah Heuser (003349645)
Alessia Tani (003350676)
Sara Zambetti (003350799)

- **Research:** Sara, Alessia, Noah
- **Preprocessing, code documentation, presentation:** Sara, Alessia
- **Classifier implementation and evaluation:** Noah

1 Introduction

The Digits-3D project addresses the problem of recognising hand-drawn digits from three-dimensional motion data captured with a LeapMotion sensor. Each sample consists of a temporal sequence describing the 3-D trajectory of a fingertip drawing a digit in mid-air, and the goal is to design a pattern recognition system capable of assigning each trajectory to one of the ten digit classes (0–9).

This task exhibits several challenges typical of gesture and time-series recognition, including variable trajectory length, differences in drawing style across subjects and sensor noise. To handle these issues, the present work first transforms raw trajectories into fixed-dimensional feature vectors through a dedicated preprocessing pipeline. On top of these features, a linear Support Vector Machine with a soft-margin is implemented and trained using the Sequential Minimal Optimization algorithm, and extended to the multiclass setting through a One-vs-One strategy. The resulting classifier is tuned by cross-validation and evaluated on a held-out test set, and the final system is exposed through a `digit_classify` function that maps a single 3-D trajectory to its predicted digit label.

2 Preprocessing

Each digit sample consists of a three-dimensional trajectory of variable length, $X = \{x_t\}_{t=0}^{T-1}$, $x_t \in \mathbb{R}^3$, which must be converted into a fixed-dimensional representation before classification. The preprocessing pipeline applies translation, arc-length normalisation, uniform resampling, velocity computation and final concatenation into a single feature vector.

To remove dependence on the absolute hand position, each trajectory is translated so that its first point lies at the origin:

$$\tilde{x}_t = x_t - x_0. \quad (1)$$

Its geometric structure is described by the cumulative arc-length, defined through

$$\ell_t = \|\tilde{x}_{t+1} - \tilde{x}_t\|_2, \quad s_t = \sum_{k=0}^{t-1} \ell_k, \quad (2)$$

where s_{T-1} is the total trajectory length. Scaling by $L = s_{T-1}$ yields a length-normalised trajectory

$$\hat{x}_t = \frac{\tilde{x}_t}{L}, \quad \hat{s}_t = \frac{s_t}{L}. \quad (3)$$

Since the number of sampled points varies across trajectories, uniform resampling is performed over the normalised arc-length domain. Given M desired output points, the sampling grid $u_k = \frac{k}{M-1}$ is used and each resampled point is obtained by linear interpolation:

$$x_k^* = (1 - \alpha) \hat{x}_t + \alpha \hat{x}_{t+1}, \quad \alpha = \frac{u_k - \hat{s}_t}{\hat{s}_{t+1} - \hat{s}_t}. \quad (4)$$

To incorporate basic dynamical information, velocities are computed as

$$v_k = x_{k+1}^* - x_k^*. \quad (5)$$

The final feature vector is constructed by concatenating all resampled positions and velocities into a single array. This representation is then used as input to the SVM-based classifier described in the next section. Some of the extracted features are inspired by the paper of Rubine [1].

3 Linear soft-margin SVM implementation

A linear Support Vector Machine (SVM) classifier with a soft-margin was implemented and trained using the Sequential Minimal Optimization (SMO) algorithm [2].

At initialization, the method receives the main SVM hyperparameters: the regularization constant C , a numerical tolerance ε , and the maximum number of SMO passes `max_passes`. During training, the model stores the Lagrange multipliers α_i , the bias term b , the training samples and their labels y_j , as well as an error cache used by the SMO heuristics.

During testing the linear kernel

$$K(x_i, x_j) = x_i^T x_j \quad (6)$$

turned out to be sufficient for the given data. There was no need for non-linear kernels, such as the Gaussian Radial Basis Function (RBF), to map the input feature vectors into higher dimensional space for them to become easily separable. Even if the results with the non-linear kernel were slightly higher, it is more likely to overfit the data.

For each training sample, the current output of the SVM is computed as

$$u_i = \sum_j \alpha_j y_j K(x_j, x_i) - b. \quad (7)$$

The corresponding prediction error for the i -th sample is defined as

$$E_i = u_i - y_i. \quad (8)$$

These quantities are used to check whether a training example violates the Karush–Kuhn–Tucker (KKT) optimality conditions and is therefore eligible for further optimisation.

The training procedure follows the SMO algorithm. At each iteration, SMO selects a pair of Lagrange multipliers, α_1 and α_2 , and updates them jointly while preserving the box constraints

$$0 \leq \alpha_i \leq C \quad (9)$$

and the linear equality constraint

$$\sum_i \alpha_i y_i = 0. \quad (10)$$

For a given index i_2 , it is first checked whether the corresponding sample violates the KKT conditions. If this is the case, the algorithm searches for a suitable index i_1 using a sequence of heuristics. The main heuristic chooses i_1 so that the absolute difference in errors, $|E_1 - E_2|$, is approximately maximised among non-bound examples ($0 < \alpha_i < C$), which typically leads to larger updates of α_2 and faster convergence. If this attempt fails, the algorithm scans all non-bound examples and, if necessary, the entire training set, following the standard SMO strategy to ensure convergence.

Given a candidate pair (α_1, α_2) and labels y_1, y_2 , the interval $[L, H]$ is then computed for the new value of α_2 . The quantity

$$\eta = K(x_1, x_1) + K(x_2, x_2) - 2K(x_1, x_2), \quad (11)$$

which corresponds to the second derivative of the objective function along the feasible direction, is evaluated. If $\eta > 0$, the unconstrained optimal α_2 is obtained in closed form and subsequently clipped to $[L, H]$; if $\eta \leq 0$, the objective function is evaluated at L and H and the better endpoint is chosen. The updated value of α_1 is then computed as

$$\alpha_1^{\text{new}} = \alpha_1 + s(\alpha_2 - \alpha_2^{\text{new}}), \quad s = y_1 y_2. \quad (12)$$

After updating the multipliers, the bias term b is recomputed using formulas that enforce the KKT conditions for the two samples, taking into account whether α_1^{new} or α_2^{new} lies strictly in $(0, C)$. Finally, the error cache is updated for all non-bound examples to keep the SMO heuristics efficient.

The high-level training loop is then implemented as follows. At initialisation, all multipliers are set to zero, the bias is set to zero, and the initial error cache is computed. The main loop alternates between passes over all training examples and passes restricted to non-bound examples only. Each pass counts how many samples lead to a successful update. If no progress is made on the non-bound set, the algorithm switches back to full passes. Training stops after a fixed number of consecutive passes without changes (controlled by `max_passes`), so that the KKT conditions are satisfied up to the chosen numerical tolerance.

After training, predictions on new data are produced using the learned model. Given a set of test samples, the method computes the kernel values between training and test points and evaluates the decision function

$$f(x) = \sum_i \alpha_i y_i K(x_i, x) - b. \quad (13)$$

The class label is then assigned according to the sign of $f(x)$, yielding outputs in $\{+1, -1\}$, consistent with the binary SVM formulation. Since only training samples with non-zero α_i (the support vectors) contribute to the sum, prediction remains computationally efficient even for relatively large training sets.

4 One-vs-One Multiclass SVM

Since the custom SVM implementation developed for this project is intrinsically binary, a strategy is required to extend it to the ten-class digit recognition task. The One-vs-One (OvO) scheme is adopted for this purpose [3]. In this approach, one binary classifier is trained for every unordered pair of classes, resulting in

$$\frac{K(K-1)}{2} = \frac{10 \cdot 9}{2} = 45 \quad (14)$$

independent SVM models when $K = 10$. Each classifier is responsible for separating only two digit classes, which reduces the complexity of every individual optimisation problem

and is well suited to the computational constraints imposed by a fully manual SMO implementation.

During training, the `OvO.SVM` class first identifies the distinct class labels contained in the vector y and constructs all pairwise combinations (c_i, c_j) . For each pair, only the samples belonging to these two classes are selected. Their labels are then mapped to a binary scheme according to

$$y_i = \begin{cases} +1, & \text{if the sample belongs to class } c_i, \\ -1, & \text{if the sample belongs to class } c_j. \end{cases} \quad (15)$$

The corresponding subset of feature vectors is then used to train a new instance of the SVM classifier, initialised with the same hyperparameters as the other models. All trained classifiers and their associated class pairs are stored internally.

Prediction is performed by evaluating all trained binary models on the input samples. For a sample x , each classifier associated with the pair (c_i, c_j) outputs a label $\hat{y}(x) \in \{-1, +1\}$. A positive output contributes a vote to class c_i , whereas a negative output contributes a vote to class c_j . If $v_k(x)$ denotes the number of votes received by class k , the final predicted label is obtained through a majority-voting rule:

$$\hat{C}(x) = \arg \max_{k \in \{0, \dots, 9\}} v_k(x). \quad (16)$$

This decision rule is deterministic, simple to implement, and compatible with the structure of the underlying binary SVM.

The OvO decomposition is advantageous in this project because the classification boundaries between digits often differ significantly from one pair to another. By allowing each model to focus on only two classes, the corresponding decision function

$$f(x) = \sum_{i=1}^n \alpha_i y_i K(x_i, x) - b \quad (17)$$

can specialise more effectively to the geometry of the pairwise separation problem. This tends to improve the overall robustness of the multiclass classifier. Moreover, training smaller binary problems alleviates the computational burden of the SMO iterations, which would become significantly more demanding in a direct multiclass formulation.

Through this architecture, the OvO framework provides a practical and theoretically grounded mechanism for extending the manually implemented SVM to the multiclass digit recognition task required in the Digits-3D project. The following sections describe how the data are prepared, how the SVM hyperparameters are selected, and how the final model is trained and evaluated.

5 Data Preparation and Train–Test Split

The data processing pipeline begins by loading all trajectory files from the dataset directory. Each CSV file is read into a list of NumPy arrays each representing a gesture trajectory, and the length of each trajectory is recorded. The class label is extracted directly from the filename and stored in the label vector y . Once all samples are loaded, the code computes the average trajectory length across the dataset.

This average length is then used to initialize the preprocessing pipeline, which standardizes all trajectories to the same number of points. By calling `pipeline.transform(samples)`, every raw sequence is resampled and normalized, producing a consistent feature set X suitable for machine learning models.

Finally, the dataset is split into training and test sets using `train_test_split`. The function randomly partitions X and y into $X_{train}, X_{test}, y_{train}, y_{test}$, ensuring that the model is evaluated on unseen data. Setting `random_state = 42` guarantees that the split is reproducible. In this way, the code prepares the entire dataset—from raw trajectories to ready-to-train splits—in a structured and reliable manner.

6 Cross-Validation and Model Training

Hyperparameter tuning for the SVM classifier is performed using 5-fold cross-validation. The code explores different values for regularization parameter C evaluating how well each setting performs on the training set.

A `KFold` splitter divides the training data into five folds, ensuring that each subset is used once as a validation set. For every value of C , the model is trained and evaluated five times—once per fold—and the validation accuracies are collected.

For each hyperparameter value, the code computes the average accuracy across the five folds and stores the results. It keeps track of the best-performing pair by comparing the mean accuracies. In the end, the value $C = 0.01$ is selected, achieving the highest average cross-validation accuracy (0.9325).

After selecting the best hyperparameters from cross-validation, the code trains the final One-vs-One SVM classifier using the full training set. The classifier is then trained with the optimal C value, and predictions are obtained for both the training data ($y_{trainpred}$) and the unseen test data ($y_{testpred}$). This final step completes the model training pipeline and produces the outputs needed to evaluate the SVM’s performance on real test samples. The resulting model is then assessed on both training and test sets, as reported in the next section.

7 Classification Results

The performance of the implemented recognition system was evaluated on both the training and the held-out test set, using the feature vectors produced by the preprocessing pipeline and the subsequent One-vs-One SVM classifier. The evaluation considers standard metrics such as precision, recall, F1-score and overall accuracy, together with the corresponding confusion matrices, which provide a detailed view of the model’s behaviour across the ten digit classes.

The results on the training set indicate that the classifier learns highly discriminative decision functions. The overall accuracy reaches 0.94, and almost all individual classes achieve precision and recall values around 0.9. Only a few isolated errors appear, mostly involving digits whose trajectories exhibit similar local geometric structures, such as the occasional confusion between digits 4 and 9 for example. The confusion matrix in Figure 1 confirms this behaviour, displaying an almost perfectly diagonal pattern.

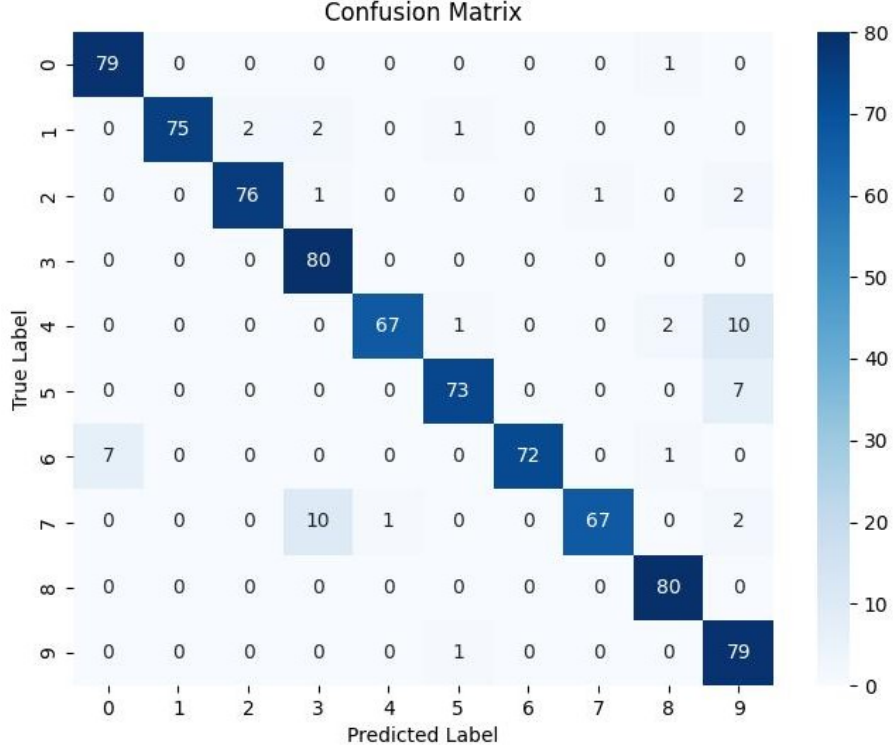


Figure 1: Confusion matrix for the training set.

On the test set, the classifier reaches an accuracy of 0.92, demonstrating strong generalisation capabilities despite considerable inter-subject variability in drawing style. Precision, recall and F1-scores remain consistently high for all digits, with macro-averaged values around 0.92. The confusion matrix in Figure 2 shows that the vast majority of samples are correctly identified. The few misclassifications that do occur reflect natural ambiguities in hand-drawn trajectories. Examples include occasional confusion between digits 4 and 9 again, which may result from similarities in curvature or incomplete stroke closure. These errors remain sparse and do not significantly affect the robustness of the system. We accept these missclassifications to hopefully be more robust for the data the classifier will be evaluated on. The RBF-kernel classifier got 1.0 accuracy on scaled training data compared to 0.94 on the test set, which is a strong sign of overfitting the training dataset.

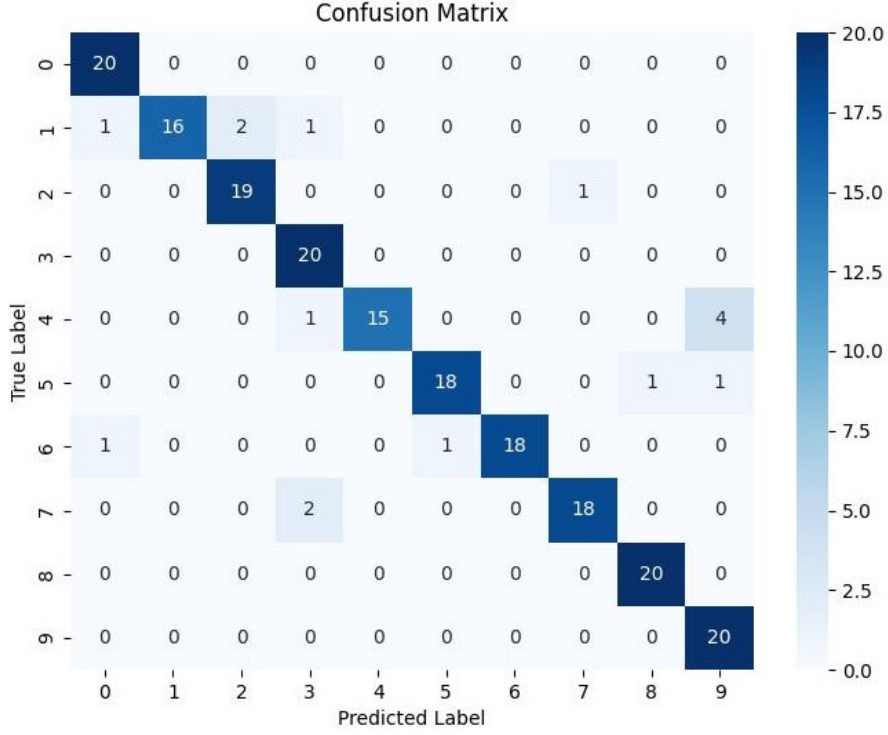


Figure 2: Confusion matrix for the test set.

Overall, the combination of arc-length normalisation, uniform resampling, velocity-based feature augmentation and SVM classification yields a highly reliable recognition framework for 3-D hand-drawn digits. The performance on both training and test sets confirms the suitability of the proposed representation and classification strategy for capturing the essential geometric and dynamical characteristics of the trajectories, resulting in a system that is both accurate and robust.

8 digit_classify Function

A dedicated function, `digit_classify`, is provided to classify a single trajectory sample using the trained model. The function expects as input one sample represented as a two-dimensional NumPy array. As a safety check, it verifies that the input is not a batch (i.e. not three-dimensional). The sample is then passed through the same preprocessing pipeline used during training, ensuring that resampling, normalisation and feature construction are fully consistent with the conditions under which the SVM was learned. After preprocessing, the resulting feature vector is fed to the final One-vs-One SVM classifier, which outputs a class label in $\{0, \dots, 9\}$. Since the classifier's `predict` method returns an array, the function extracts and returns the single predicted label.

For validation, the function was tested on randomly selected samples from the dataset by comparing the predicted label with the corresponding ground-truth class. These tests provide a sanity check that the entire preprocessing and classification pipeline operates consistently even when applied to individual trajectories.

References

- [1] D. Rubine, “Specifying gestures by example,” *Computer Graphics (SIGGRAPH '91 Proceedings)*, vol. 25, no. 4, pp. 329–337, July 1991.
- [2] J. C. Platt, “Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines,” Technical Report MSR-TR-98-14, Microsoft Research, 1998.
- [3] C.-W. Hsu and C.-J. Lin, “A comparison of methods for multiclass support vector machines,” *IEEE Transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, 2002.