University of Waterloo

Multistroke Character Recognition Using Orthogonal Polynomial Representations

by

Arun Cheriakara Joseph

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Abstract

This thesis studies stroke grouping for online word-level handwriting recognition of Latin letters and digits using orthogonal polynomial representations of pen strokes. A word arrives as an ordered sequence of pen-down strokes, and the system has to decide which strokes belong to which character before it can decide what each character is. At the word level the problem is harder than for isolated characters: the right grouping of strokes depends on what the characters turn out to be, and the right characters depend on how the strokes are grouped. Most existing systems commit to one segmentation and use whatever that segmentation outputs, which can lead to wrong results. The difficulty is sharpened by characters drawn with multiple strokes, by variation in stroke order between writers, and by several letter pairs and letter/digit pairs that share the same shape.

This thesis describes an online word-level recognition pipeline built on orthogonal polynomial representations of multistroke characters. Each pen stroke is re-parameterized by arc length, and its coefficients are projected onto an orthogonal Legendre basis of degree eleven, giving a fixed-length coefficient vector per stroke. For multistroke characters, the per-stroke vectors are concatenated into a single feature vector. Because all strokes in a character are normalized together against a shared bounding box, this block-concatenated representation captures the relative position and scale of the strokes within the character, but it does not directly encode every pairwise relationship between strokes. A probabilistic gap model generates up to six candidate groupings per word, and each candidate character group is normalized in a common bounding box before projection. The resulting vectors are matched against a reference database of 76,428 samples across 62 character labels, organized into 3,237 classes. Classification runs in two stages: a centroid-and-radius heuristic prunes the candidate pool to fifty classes, and a label-pooled k-nearest-neighbour stage then ranks the seven closest samples per label by distance to the convex hull of those samples. The pipeline is evaluated on the UniPen word collection drawn from the 62-character Latin-plus-digits alphabet.

Keywords: online handwriting recognition, multi-stroke recognition, orthogonal polynomial representation, Legendre coefficients, convex-hull KNN, trace grouping, UniPen.

Acknowledgements

First and foremost I would like to express my deepest gratitude and respect to my supervisor, Dr. Stephen M. Watt. His vast knowledge of online handwriting recognition, his patient and constant guidance through every stage of this thesis, and the calm manner in which he met every question I brought to him are what made this work possible. Without any of these, this thesis would not have been completed. There were many points where the problem in front of me seemed larger than I knew how to handle, and each time he sat with me, listened, and helped me see the next step without ever making me feel rushed or behind. The standard he sets for clarity of thought, and the steadiness with which he holds it, are things that are hard to find in books and will stay with me long after this thesis is done.

I also wish to thank the University of Waterloo for hosting this work and for providing the environment, resources, and community that made it possible to pursue a thesis of this scope. I am grateful to the readers of this thesis for the time they put into examining it and for the feedback that improved the final document.

Last, I wish to thank my family for their constant love and support across the years of this work. To my parents, who always encouraged the pursuit of difficult problems and never asked me to stop, thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Online handwriting recognition differs from recognition of static images of handwriting. A static image gives only the final inked shape, while online recognition has access to the timed sequence of stroke coordinates as they are written. Digital ink is now common on tablets, laptops, and smartphones, and these devices are used for applications such as note-taking, software development, and mathematical work. This captured trajectory is the basis for online handwriting recognition as set out in the survey by Plamondon and Srihari (2000) [1]. InkML has emerged as the primary format for the traces created by digital ink [2]. Even with a standard trace format such as InkML, recognizing the underlying characters of the trace remains challenging.

The main goal of this thesis is straightforward but hard to accomplish: to decide which strokes belong to which letters in a written word, working in an orthogonal-polynomial coefficient representation of the strokes. In handwritten words, individual letters are often not easily separable. Letters may be drawn with one stroke, while other letters may be composed of multiple strokes. Many adjacent letters will make contact or overlap, making it possible for a stroke to visually appear to be associated with a neighbouring letter while still being assigned to the correct letter by a human reader. Casey and Lecolinet (1996) analyzed character segmentation techniques, and they concluded that segmentation errors represent a significant portion of total OCR error rates [3].

Therefore, instead of using a technique that commits to a specific split too early in the process, the proposed method uses an alternative strategy where multiple potential letter candidates are generated by grouping nearby strokes together. Each candidate is normalized, mapped into a polynomial feature space as described in the paper by Char and Watt (2007) [4], and compared against a previously developed class database and scored. Due to ambiguity during the recognition process, the highest-scoring sequence of

letters can be determined rather than committing to an earlier decision, which can lead to propagation of errors.

1.2 Why this problem matters

Handwriting recognition has been studied extensively. Current applications require recognisers that can function accurately across varying writing styles and with sufficient speed for interactive use. Zanibbi and Blostein (2012) documented common characteristics of mathematical handwriting and identified that many math symbols consist of multiple strokes that are close together and often appear very similar visually [5].

Char and Watt (2007) developed orthogonal-series representations for handwritten mathematical symbols [4]. These were further developed in Mazalov's (2013) paper on classification by functional approximation [6]. The most direct classifier predecessor is the distance-based classification of handwritten symbols developed by Golubitsky and Watt (2010) in this same coefficient space [7]. Alvandi and Watt (2020) demonstrated that Legendre-Sobolev coefficients provide a form of compact shape representation of handwritten mathematical symbols that are ideal for comparing distances [8].

This thesis builds upon that line of work, which is related to the use of recognizers to classify handwritten words based on their content, specifically, how to handle ambiguous grouping at the word level prior to making final classifications. It develops a pipeline that examines several possible ways that letters may be grouped within a word before performing classification, and reports its recognition accuracy on the Latin-plus-digits UniPen word collection. A direct comparison against a single-segmentation baseline is left to future work. Figure 1.1 shows a handwritten word as it is stored in InkML, and Figure 1.2 shows six sample letters from the reference database that the recogniser matches against.

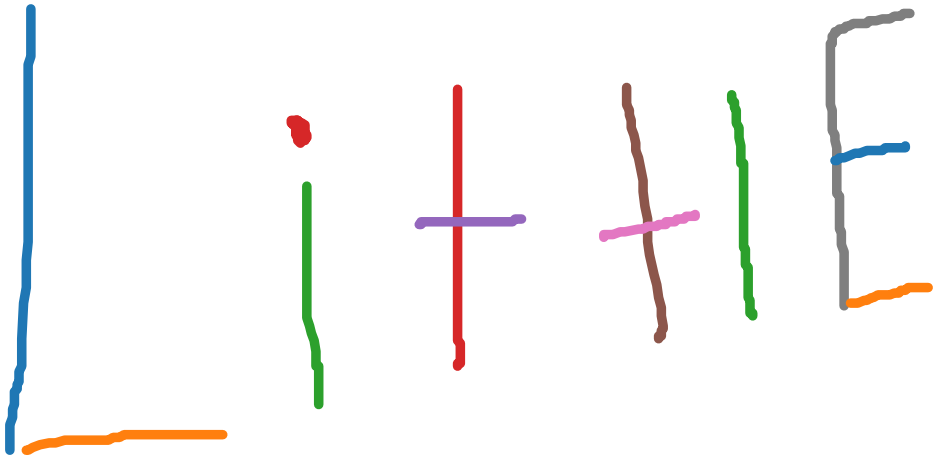

Figure 1.1: A handwritten word stored as InkML. One colour per pen stroke.

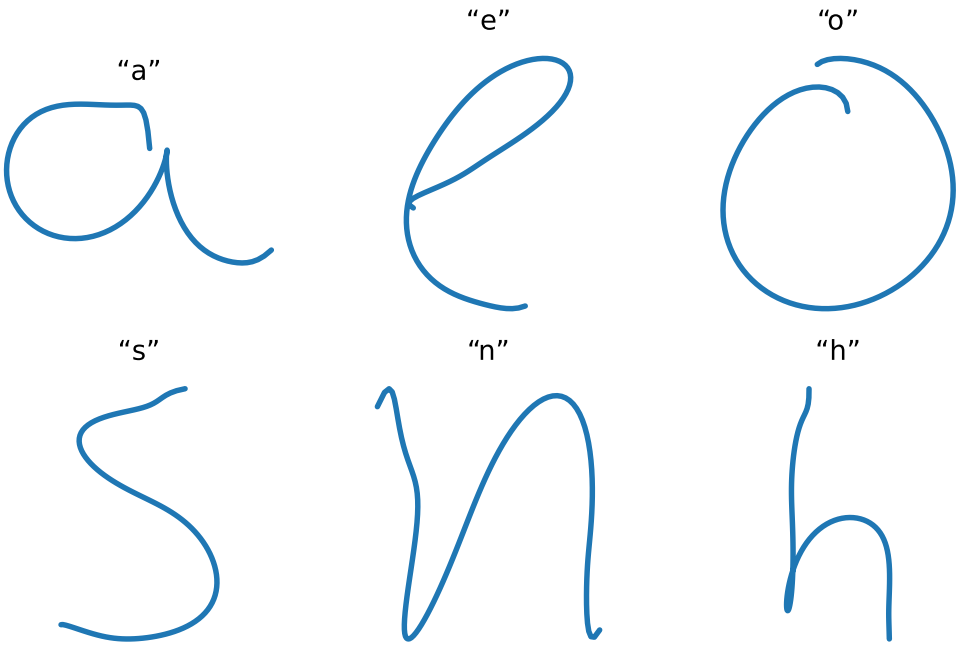

Figure 1.2: Six sample InkML letters from the reference database, reconstructed from their stored coefficients.

1.3 Main contributions

The orthogonal polynomial representation of strokes, the distance-based classification of the resulting coefficient vectors, and the use of convex-hull nearest-neighbour scoring are inherited from earlier work on isolated symbol recognition. The contributions specific to this thesis sit on top of that representation: word-level trace grouping, the way candidate segmentations are generated and scored, the database construction choices, and the end-to-end evaluation on Latin-plus-digits UniPen words, all carried out in the orthogonal polynomial basis representation. The main contributions of this thesis are:

1. Word-level recognition using candidate character groups derived from stroke groupings, rather than a fixed grouping.
2. A normalization and feature-extraction workflow using orthogonal polynomial coefficients to represent letters.
3. A distance-based classification pipeline for multi-stroke symbols consisting of a class reduction step followed by k-nearest-neighbour matching.
4. A geometric refinement and confidence-scoring stage based on convex-hull distance for ranking reliability.
5. An end-to-end implementation that integrates candidate generation, per-character scoring, and word-level selection.

1.4 Why a distance-based approach

Modern online handwriting recognition is often done with deep neural networks, but these need thousands of labelled examples per class to train. Instead, the nearest convex hull of per-class k-nearest neighbours classifier, used in this thesis, can be built with as few as seven samples per class. The goal is not to beat deep networks on accuracy, but to study the stroke-grouping problem with a method that needs very little training data.

1.5 Thesis organization

The rest of the thesis is organized as follows:

- **Chapter 2** presents prior work, including online ink basics, InkML, and core mathematical tools.
- **Chapter 3** describes normalization for candidate letters grouped from traces.
- **Chapter 4** presents polynomial feature construction.
- **Chapter 5** defines the class database and stored reference structure.
- **Chapter 6** presents the full recognition pipeline: probabilistic trace grouping, k-nearest-neighbour classification, and convex-hull confidence ranking.
- **Chapter 7** covers experimental design and evaluation setup.
- **Chapter 8** reports results and error analysis.
- **Chapter 9** discusses limitations and future work.
- **Chapter 10** concludes the thesis.

Chapter 2

Preliminaries and prior work

2.1 Digital ink representation

Online handwriting systems are primarily composed of pen-based systems. These are based on a digital pen that captures the user's hand movements, creating a trace. This trace is then digitized into a sequence of coordinate points in both the x and y directions as well as in time (x, y, t). Several datasets of digital ink traces have been collected, the most prominent of which is the UNIPEN dataset. The primary goal of UNIPEN was to create a standardized method for exchanging handwritten data captured by online applications and to provide benchmarks for recognizing those exchanges. Some original source websites are no longer available, but the data is available in various archive releases, including the Zenodo-hosted distribution referenced here [9]. In reality, most vendors continue to capture digital writing as images, which removes valuable information such as stroke boundary, order of strokes, timing, pressure, and additional pen attributes.

This thesis will use the Ink Markup Language (InkML) as the input format. InkML is a W3C-recommended markup language designed specifically to describe digital ink. One of the major benefits of InkML when applied to the problem of online recognition is that traces are described explicitly as an ordered list of sampled point locations. InkML only requires the use of the two coordinates (x and y). However, it allows for any number of additional optional coordinates (such as time, pressure, device type, etc.) to be associated with each sampled location. Due to its flexible nature, InkML lends itself well to developing recognition algorithms that take advantage of the spatial structure inherent within the individual strokes and the metadata associated with those strokes. Additionally, InkML facilitates cross-platform and cross-database compatibility.

The traces in the InkML input are captured as one or more pen traces:

$$\mathcal{T} = (T_1, T_2, \ldots, T_m),$$

where each trace T_i is an ordered sequence of sampled points

$$T_i = \big((x_{i1}, y_{i1}, t_{i1}), \ldots, (x_{in_i}, y_{in_i}, t_{in_i})\big).$$

This mathematical view is the starting point for the rest of the thesis. The W3C Ink Markup Language (InkML) is the standard format used in this thesis for digital ink data representation and parsing. It provides a practical and standards-based foundation: it keeps the pipeline grounded in real online ink data (strokes, order, optional channels), while allowing the recognition system to operate on a compact, normalized representation derived from that ink.

2.2 Legendre polynomials

In online handwriting recognition, strokes are modeled as planar parametric curves whose coordinate functions can be approximated by truncated orthogonal polynomial series. These coefficients then form compact shape descriptors suitable for classification. This subsection provides an introduction to the Legendre basis used to develop these approximations and relates it to the Legendre–Sobolev basis described in prior work.

The Legendre polynomials P_n are orthogonal on $[-1, 1]$ with respect to the L^2 inner product

$$\langle f, g\rangle_L := \int_{-1}^{1} f(\lambda)\, g(\lambda)\, d\lambda.$$

The orthonormal basis is

$$\phi_n(s) = \sqrt{\frac{2n+1}{2}}\, P_n(s), \quad s \in [-1, 1],$$

so that $\langle \phi_i, \phi_j\rangle_L = \delta_{ij}$.

2.3 Legendre–Sobolev polynomials

The Legendre–Sobolev inner product adds a derivative term

$$\langle f, g\rangle_{LS} := \int_{-1}^{1} f(\lambda)\, g(\lambda)\, d\lambda + \mu \int_{-1}^{1} f'(\lambda)\, g'(\lambda)\, d\lambda,$$

where $\mu \in \mathbb{R}_{>0}$. The corresponding orthogonal polynomials S_n^{μ} (Legendre–Sobolev) are obtained by Gram–Schmidt from the monomials. This inner product penalizes high-frequency oscillations and, in handwriting recognition, tends to improve detection rates compared to using Legendre alone [8].

2.4 Arc-length parameterization

Each trace is treated as a planar parametric curve

$$\gamma(s) = (x(s), y(s)).$$

These traces are parameterized by normalized arc length $s \in [-1, 1]$, taken along the piecewise-linear curve through the sampled points rather than along any smoothed or fitted curve. Arc-length parameterization is used to reduce the impact of nonuniform sampling density and to reduce sensitivity to writing speed variations. As a result, traces with similar trace geometry but different sampling rates are parameterized in the same domain and the Legendre polynomial projection can be applied.

2.5 Orthogonal polynomial approximation

After parameterization, each coordinate function is approximated in an orthogonal basis:

$$x(s) \approx \sum_{j=0}^{d} a_j \phi_j(s), \qquad y(s) \approx \sum_{j=0}^{d} b_j \phi_j(s),$$

with degree d fixed (in this thesis, $d = 11$). The basis functions ϕ_j are the standard Legendre polynomials P_j, orthogonal on $[-1, 1]$ with $\langle \phi_i, \phi_j \rangle_L = (2/(2j+1))\,\delta_{ij}$, aligned with the arc-length parameter domain. The coefficients are obtained by projection:

$$a_j = \langle x, \phi_j \rangle = \int_{-1}^{1} x(s)\,\phi_j(s)\,ds, \quad b_j = \langle y, \phi_j \rangle = \int_{-1}^{1} y(s)\,\phi_j(s)\,ds.$$

The coefficient vectors

$$\mathbf{a} = (a_0, \ldots, a_d), \qquad \mathbf{b} = (b_0, \ldots, b_d)$$

form a compact shape descriptor for each trace.

This follows the orthogonal-series approach used in online handwriting recognition, where functional coefficients are a device-independent alternative to raw point sequences and support stable distance computations.

2.6 Prior work

The pipeline in this thesis sits next to several research areas at once. The most direct ancestry is orthogonal-series handwriting recognition built around the Legendre–Sobolev representation. Around that sit the older online-recognition methods (dynamic time warping, hidden Markov models, prototype classifiers), the deep-learning systems that replaced them, math handwriting work from other groups, the wider literature on shape descriptors, distance-based classifiers, and the datasets and segmentation methods used for evaluation. Each subsection below covers one of these areas and says how the pipeline here connects to it.

2.6.1 Orthogonal-series handwriting recognition

The work developed in this thesis is based upon a succession of works published by the research group led by Stephen M. Watt. These works share common principles of treating a handwritten symbol as a parametric curve; projecting the coordinate functions onto a basis composed of low-degree orthogonal polynomials; and performing symbol identification

through a process of comparing distances between the representations of each symbol within the coefficient space established after projection. Each of the papers presented below has remained consistent with these principles and has further refined one aspect or another of them.

In their initial contribution, Char and Watt (2007) [4] modeled the $x(t)$ and $y(t)$ coordinate functions of a symbol using truncated Chebyshev series of approximately tenth order. In addition, they demonstrated that when samples of the same character were created by different writers, those samples would form clusters in the coefficient space. This contributed to the central argument presented by the authors, which is that a small number of polynomial coefficients contains sufficient information about the shape characteristics of a symbol to allow for its successful classification. Additionally, Char and Watt identified two open problems (computation during online input, processing of multistroke symbols) to be addressed in subsequent contributions.

Golubitsky and Watt (2008) [10] replaced the use of the Chebyshev basis with the use of the Legendre basis, whose flat weighting functions allow the integration required to compute coefficients to occur incrementally as the symbol is being drawn. Additionally, Golubitsky and Watt reformulated the calculation of coefficients as a recovery of Hausdorff moments. The same paper provided an additional rationale for why arc length is used instead of time as a parameter in modeling curves (arc length provides a more invariant definition).

A final extension of the above ideas was explored by Golubitsky and Watt [11] in their 2009 ICDAR contribution. The paper generalized the representation to include multistroke symbols by connecting successive strokes with line segments whose slope was fixed at degree twelve. The Legendre–Sobolev inner product (which includes a Legendre inner product along with a derivative term) was used. An ensemble of pairwise linear Support Vector Machines (SVMs) was then used to classify the symbols. The reasoning was that the linear homotopy between two samples of the same class produces another sample of that class, so the class is convex and pairs of classes are linearly separable.

Distance-based classification reached a good state in Golubitsky and Watt (2010) [7], which is the single closest predecessor of the pipeline here. They evaluated elastic matching, Euclidean distance, Manhattan distance and the Euclidean distance to the convex hull of the k nearest neighbours on a dataset of 50,703 isolated math symbols representing 242 classes; the convex hull technique scored 97.5%. Golubitsky and Watt also introduced

the concept of a two-step procedure to select candidates with fast Manhattan distances, followed by the use of slower convex hull distances to break ties. All of these techniques have been used again in the per-character classifier of this thesis.

The other two additions were made by Mazalov and Watt. In their 2012 studies [12], they extended the isolated-symbol recognizer to recognize strings of rotated characters together and to include an absolute size measurement to separate punctuation from letters [13]. Their research was pulled together by Mazalov in his 2013 paper [6]. Alvandi and Watt (2019) [14] took the Legendre–Sobolev approximation as a matrix transformation, acting on the moment vector so that geometric operations could be applied to the resulting polynomials in an orthogonal basis.

All of this prior work is isolated-symbol work. The benchmarks reported above (the 97.5% of [7], the 3.75% rotated error of [12], the size-aware results of [13], the 11–20% multistroke error of [11]) all assume the segmentation is given. The recognizer is handed one character at a time, already cut out, and asked only to assign a label. This thesis extends the convex-hull-of-k-nearest-neighbours classifier described in [7] as the per-character component and places it inside a larger word-level processing pipeline where data segmentation needs to take place as well. An input stream of stroke segments is grouped into candidate trace groups, then each group is normalized and processed independently as if it represented an individual character. The scores produced by the per-character classifiers are combined into a single string by selecting the candidate word with the highest mean per-character confidence. The polynomial representation, Legendre–Sobolev distance, convex-hull tie-breaker, and proof of convexity are all adapted; stroke grouping, candidate lattice search, and word-level scoring represent new components that are added on top of this prior work.

2.6.2 Classical online handwriting recognition before deep learning

Prior to deep sequence models taking over, online handwriting recognition was dominated by three main families of algorithms: elastic matching using dynamic time warping, hidden Markov models adapted from speech, and prototype or kernel classifiers on per-character feature vectors.

The dynamic programming origins date back to Sakoe and Chiba (1978) [15], which formalized a symmetric DP-matching algorithm for spoken word recognition. Tappert [16]

extended the elastic matching concept to online cursive script by matching words against per-letter prototypes and generating the letters of the word as the concatenation of the individual prototypes most similar to the trace. The thesis maintains the segmentation-then classification logic at the word level; however, the distance metric is now calculated in the coefficient space versus using dynamic programming to align the original samples.

Hidden Markov models gained prominence in the late 1990s. Starner and Makhoul (1994) [17] adapted the BBN BYBLOS speech system to online cursive sentences. This approach resulted in writer-dependent performance metrics of 1.1%. A time delay neural network was placed in front of the previously mentioned HMM with dictionary constraints. Schenkel, Guyon, and Henderson (1995) [18] achieved approximately 80% accurate words on writer-independent data. Similarly, Liwicki and Bunke (2006) [19] employed a comparable model to recognize written notes produced via an electronic whiteboard. The LeRec System developed by Bengio, LeCun, Nohl, and Burges (1995) [20] used both replication of a convolutional network along with joint training of an HMM. The thesis uses a portion of these system architectures - specifically, the outermost decomposition into character-based scoring, followed by word-level classification. However, unlike the HMM- and TDNN-based systems above, the scoring function will be non-parametric, using KNN as opposed to a learned network or HMM.

Prototype methods sit alongside the HMM line. Bahlmann, Haasdonk, and Burkhardt (2002) [21] proposed the Gaussian DTW kernel such that Support Vector Machines may take advantage of variable-length pen traces. Their CSDTW system utilized clustering and HMM-like models within a single feature space [22]. The Hierarchical Clustering KNN pipeline described by Vuori et al. (2002) [23] has a hierarchical clustering component that produces per-class prototype sets, while DTW generates the similarities. This thesis utilizes a subset of this architecture, the prototype-KNN piece, and substitutes Euclidean distance in Legendre–Sobolev coefficient space for DTW. Additionally, it replaces the confidence generated from selecting the nearest prototype with a convex hull distance that provides a rejection signal. The survey by Plamondon and Srihari (2000) covers the period in detail [1].

2.6.3 Deep sequence learning for online handwriting and math expressions

The next group of related work skips per-character classifiers entirely and treats the stroke trace as a sequence to be transcribed end to end. Graves and co-authors introduced connectionist temporal classification (CTC) in 2006 [24], which removes the need for pre-segmented training data. Three years later, the same group combined CTC with bidirectional LSTM and reached 79.7% word accuracy on IAM-OnDB, beating the HMM baselines that had been dominant [25]. The multidimensional LSTM line [26] carried the same recurrent machinery over to offline pixel input, and Graves' monograph pulls the framework together [27]. Frinken and Bunke (2015) stacked deeper BLSTM layers on the IAM offline benchmark [28], and Carbune and colleagues (2020) replaced Google's segment-and-decode online handwriting product with a 102-language BLSTM-CTC system that uses a Bézier-curve input encoding [29]. These systems illustrate that when the network and training set are sufficient in size, features no longer have to be manually engineered. The thesis follows another direction: features were defined analytically based upon the geometric properties of curves, and the classifier was non-parametric. It limited the amount of required data and provided a per-class decision process that could be audited.

Online handwritten math expression recognition went through a parallel evolution, from grammar-based parsers to attention-based encoder-decoder networks [30, 31, 32] and on to transformer decoders [33, 34]. Jungo and colleagues (2023) treated character segmentation as an assignment problem with learned queries in a transformer decoder [35], and Fadeeva and colleagues (2024) encoded ink for vision-language models and matched specialized online recognizers on IAM-OnDB [36]. Le and Nakagawa (2016) predates this wave and uses a stochastic context-free grammar over a CYK lattice with SVMs on structural features [37]. Distance to a class hull in a fixed feature space is a smaller, more interpretable model class than a transformer decoder, but it gives an explicit per-character confidence, an interpretable feature space, and a training cost that comes down to centroid computation rather than gradient descent on millions of parameters.

2.6.4 Math handwriting recognition from other groups

Mathematical handwriting recognition is closely related to word recognition but not the same problem. The symbol set is larger, the layout is two-dimensional, and the output is

a structured tree rather than a string. Zanibbi and Blostein (2012) survey the area [5], and Tapia and Rojas (2007) cover the online side specifically [38]. CROHME has been the public benchmark since 2011, with a retrospective by Mouchère and colleagues (2016) [39] and the 2019 edition organised by Mahdavi, Zanibbi, and Mouchère [40]. Early pen-based math interfaces include MathPad^2 by LaViola and Zeleznik (2004) [41, 42], the equation editor of Smithies and colleagues (1999) [43], and the E-Chalk system of Tapia and Rojas (2003) [44].

Hu and Zanibbi developed several component-based approaches that break the problem into separate components: an HMM symbol classifier [45], an AdaBoost-based segmenter using shape-context features [46], a line-of-sight stroke graph that uses stroke convex hulls as the visibility primitive [47], and an MST parser over that graph [48]. Davila, Ludi, and Zanibbi (2014) add an offline-feature symbol classifier [49], and Hu's RIT paper pulls the line of work together [50]. The MathBrush system, developed by MacLean and Labahn, is the most similar in design choices: their 2013 paper presents a fuzzy relational context-free grammar [51], and the 2015 paper replaces the fuzzy formalism with a Bayesian scoring model over a parse forest [52]. Both MathBrush papers carry multiple parse hypotheses through the system rather than committing to a single segmentation early, which is the same design choice made here.

2.6.5 Shape descriptors and orthogonal-series representations of curves

Representing a simple planar form using numerical parameters invariant under translation, scaling, or rotation has been done for many years. One way to do this is to use the image itself. Hu's invariant moments (1962) [53] are one example. Another way is to use the edge curve. In 1972, Zahn and Roskies [54] showed how to represent a closed curve with a single parameter. They represented each closed curve by parameterizing it by arc length, expanding the total angular change around the curve as a Fourier series, and then applying the resultant descriptors to character and machine recognition. It was these representations that were compared in Teh and Chin (1988). Teh and Chin evaluated several types of moment families, including Legendre, Zernike, and others, in terms of their ability to represent shapes, their sensitivity to additive random error, and their information redundancy [55]. This thesis follows the same logic as Teh and Chin, but applies this logic

to a curve that had first been parameterized by arc length over $[-1, 1]$, and then treats both x and y coordinate functions as functions defined on $[-1, 1]$. For each stroke in the handwritten document, a degree-11 Legendre projection generates 12 coefficients for each of the x and y coordinates.

There are also a number of closely related studies that have been carried out in the image domain rather than along a parameterized curve. Bres et al. (2006) [56] applied the Hermite Transform to handwritten documents. Belongie et al. (2002) [57] introduced Shape Contexts from log-polar histograms of contours. Feng et al. (2010) [58] developed Affine Integral Signatures, where the value represents an integral invariant under the action of the affine group. The design spaces for all three methods differ from those used for the Legendre–Sobolev expansions used in this study: image-domain filtering, point-set matching, and affine-invariant integration versus arc-length parameterization into an orthogonal polynomial basis.

2.6.6 Distance-based and instance-based classification

Aha, Kibler, and Albert (1991) [59] defined an instance-based method of classification, an algorithm that determines a query's class based on how similar that query is to all of the training instances for that class. Wilson and Martinez (2000) [60] surveyed many methods that reduce the storage and noise problems inherent to these kinds of classifiers. The centroid-based class reduction part of this thesis follows exactly the same logic: Each class is represented by a centroid and a ninetieth percentile radius. The search is limited to those classes whose centroids are sufficiently close to the query.

The most direct external precedent for the per-character classifier is Vincent and Bengio [61], who noticed that when using the traditional k-nearest-neighbour classifier, there is a problem where the boundary of each class's manifold, the local margins for the classification decisions are very poor. To solve this, they proposed approximating each class locally as the convex hull of its k nearest neighbours to the query. Then the query would be assigned to the class for which the distance from the query to its convex hull was smallest. When tested on USPS and MNIST, this convex-distance classifier gave results equivalent to a Gaussian kernel Support Vector Machine (SVM), but handled multiclass problems and required no training beyond storing examples. This thesis makes use of the Vincent-Bengio approximation directly. After centroid-based class reduction has narrowed the

candidate set, samples from the surviving labels are pooled into a single search pool. For each surviving label, the method takes the k nearest samples under the Sobolev-weighted metric, forms their convex hull in coefficient space, and ranks the label by point-to-hull distance. In addition, two modifications were made: the point-to-hull distance is transformed into a confidence value that allows for rejection and for selecting which words should be included in a particular segment. The per-character hull distances are then combined into a word-level decision by selecting the candidate word with the highest mean per-character confidence.

2.6.7 Datasets and segmentation theory

UNIPEN is the standard benchmark for online handwriting. It was set up in 1994 by Guyon and colleagues on behalf of IAPR Technical Committee 11 [62, 9]. The test set used in this thesis is a filtered subset of the UNIPEN word data, restricted to Latin letters and digits. Filtering is needed because UNIPEN labels are known to contain errors; Vuurpijl and colleagues (2004) report on a semi-automated verification of the UNIPEN devset in which earlier estimates put the training-set error rate at roughly four percent [63]. Two design choices in this thesis follow from that: a rejection threshold on low-confidence predictions, and a superclass-corrected accuracy metric that collapses confused pairs.

Casey and Lecolinet (1996) give a clean classification of word-level segmentation: dissection, recognition-based segmentation, and holistic recognition [3]. The pipeline here is recognition-based in their sense. Stroke grouping produces a lattice of candidate character segmentations, each candidate is scored by the convex-hull classifier, and the final word is selected by argmax over candidate words ranked by mean per-character confidence. Bunke (2003) names the underlying obstacle as Sayre's paradox: characters cannot be recognised without segmentation, and cannot be segmented reliably without recognition [64]. The recognition-based design here gets around this by holding off on the segmentation decision until every candidate has been scored. Manmatha and Rothfeder (2005) provide the gap-metric baseline for spacing-based word segmentation [65].

2.7 Positioning of this thesis

Relative to prior work, this thesis contributes an integrated word-level recognition pipeline that combines:

- **Probabilistic trace-grouping:** generation of multiple candidate stroke groupings via gap-based join probabilities (certain joins, ambiguous, and separate gaps), rather than a single segmentation.
- **Per-group normalization and Legendre projection:** arc-length parameterization, degree-11 Legendre coefficients for each candidate character group.
- **Two-stage classification:** centroid-based class reduction to reduce the number of candidate classes, then KNN refinement over database samples (not centroids) for each pruned class.
- **Convex-hull ranking and confidence:** final class selection by distance to the convex hull of the k nearest samples, with confidence derived from hull distance and class radius.

The pipeline is designed to preserve geometric structure (e.g. stroke-order and stroke-direction handling), handle ambiguous groupings through multiple candidates, and provide interpretable confidence values for each character decision.

Chapter 3

Data and normalization

3.1 Input and output formulation

This section describes the input representation and how the data moves through the pipeline. Offline database construction is presented separately in Chapter 5.

Input representation

A word sample parsed from an InkML file (Section 2.1) is represented as an ordered set of N traces:

$$\mathcal{T} = (t_1, t_2, \ldots, t_N),$$

where each trace/stroke corresponds to a single pen-down segment, which is the continuous path traced from the moment the pen contacts the writing surface to the moment it is lifted. Each trace has its own sequence of coordinates:

$$t_i = \big((x_{i1}, y_{i1}),\ (x_{i2}, y_{i2}),\ \ldots,\ (x_{in_i}, y_{in_i})\big).$$

The number of points n_i differs across strokes and it depends on the length of the stroke and the sampling rate of the capture device [1]. Each set of x–y coordinate values represents the sequence of when a user drew a stroke: (x_{i1}, y_{i1}) is the point at which the pen first touched down, and (x_{in_i}, y_{in_i}) is the point at which the pen was lifted.

Only the x–y coordinates are used for the geometric processing described in this thesis. Optional InkML channels such as time, pressure, or pen tilt are not used in the current pipeline. This is deliberate because the recognition pipeline is based on the shape geometry of strokes rather than other characteristics such as timing or force, following the functional approximation tradition established in [4].

The segmentation problem

A central challenge in word-level online handwriting recognition is that strokes do not map one-to-one to characters [3, 11]. The relationship between strokes and characters is many-to-many:

- A single character may consist of multiple strokes. For example, the letter *i* has a body stroke and a dot, the letter *t* has a vertical stroke and a crossbar, and the equals sign has two horizontal bars.
- Some characters are drawn in a single continuous stroke, such as the letter *o* or the digit *0*.
- Adjacent characters in a word may be written close together, with small gaps that may make the strokes part of the same letter.

The system must first determine what groups of strokes make up different characters using the raw trace set $\mathcal{T}$ before it can classify those characters. The task of determining the groups of strokes is referred to as grouping, and the errors made in the grouping process directly affect how well the characters are classified. If a stroke is incorrectly grouped with another stroke, then both characters will be misclassified.

A naive approach would be to pick a single best grouping and commit to it. This is fragile: when two grouping hypotheses look about equally plausible, a single-best strategy throws away the alternative that may turn out to be correct after classification [16]. Therefore, the approach used in this thesis was to create a number of possible candidate grouping hypotheses and run each candidate through the complete recognition process. Once all classification results have been obtained, the best grouping hypothesis can be determined based on the resulting classification scores [66].

Candidate segmentations

The system generates C candidates from the trace set:

$$\mathcal{S}^{(c)} = \{g_1^{(c)},\ g_2^{(c)},\ \ldots,\ g_{M_c}^{(c)}\}, \qquad c = 1, \ldots, C,$$

where:

- C is the number of segmentation candidates (controlled by a parameter; default 6),

- M_c is the number of character groups in candidate c,
- each group $g_j^{(c)} \subseteq \{1, \ldots, N\}$ is a set of stroke indices predicted to form one character.

Groups in each candidate are disjoint and cover all strokes:

$$\bigcup_{j=1}^{M_c} g_j^{(c)} = \{1, \ldots, N\}, \qquad g_j^{(c)} \cap g_k^{(c)} = \emptyset \text{ for } j \neq k.$$

Each candidate is one hypothesis for how the trace set should be partitioned into characters. Candidates may differ in the number of character groups M_c (e.g., one candidate may treat two adjacent strokes as a single two-stroke character, while another treats them as two separate one-stroke characters) and in which strokes are assigned to which groups.

This preserves the grouping and carries it forward into normalization, feature extraction, and classification, rather than throwing away alternate hypotheses. The grouping step itself is described in Section 6.1, which covers the probabilistic gap analysis between strokes.

Per-group processing

For each candidate c and each group $g_j^{(c)}$, the grouped trace object is defined as

$$G_j^{(c)} = \{\, t_i \mid i \in g_j^{(c)} \,\}.$$

This is the fundamental unit of processing in the normalization and feature extraction stages. All geometric operations described in the remainder of this chapter, bounding box normalization (Section 3.2), arc-length parameterization (Section 3.3), and metadata extraction (Section 3.5) are applied to each grouped trace object independently.

Working at the group level rather than the stroke level matters because of spatial relationships. Multi-stroke characters such as A, i, or $=$ have to be normalized as a unit so that the spatial relationships between their component strokes (e.g., the position of the dot above the body of i, or the angle at which the two strokes of A meet) are preserved [11]. Normalizing each stroke on its own would destroy these relationships and make multi-stroke characters indistinguishable from unrelated collections of strokes.

This design also means that the normalization result depends on the candidate grouping: different grouping hypotheses can produce different bounding boxes and therefore different normalized coordinates for the same raw strokes. This is intentional. It lets the pipeline evaluate each grouping hypothesis on its own geometric terms.

Output of the normalization stage

The output of the normalization stage is, for each candidate, a sequence of normalized character groups:

$$\widetilde{\mathcal{S}}^{(c)} = \left(\widetilde{G}_1^{(c)},\ \widetilde{G}_2^{(c)},\ \ldots,\ \widetilde{G}_{M_c}^{(c)}\right),$$

together with the metadata per group, which are described in later chapters:

- **Stroke endpoints**: the starting and ending positions of each stroke in normalized coordinates, retained for use in the classification stage (Section 6.2).
- **Raw bounding-box size**: $\sigma_{\text{raw}} = \max(w, h)$ before normalization, retained for use in the classification stage (Section 6.3).

These normalized groups are then projected onto the Legendre polynomial basis in Chapter 4 to produce the coefficient vectors used for classification.

Pipeline summary

The end-to-end flow for a single input word is:

1. Parse InkML to obtain the trace set $\mathcal{T}$.
2. Generate C candidate segmentations from the trace set (Section 6.1).
3. For each candidate, normalize every character group into a common coordinate frame.
4. Project each normalized group onto the Legendre basis to obtain coefficient vectors (Chapter 4).
5. Reduce the set of candidate classes via centroid distance, then refine with k-nearest-neighbour search and convex hull ranking to determine the best-matching class for each character group (Sections 6.2–6.3).
6. Assemble per-character predictions into candidate words and select the highest-scoring prediction as the final output (Chapter 6).

Each candidate segmentation is processed independently through steps 3–6, and the final output is the highest-ranked candidate word across all segmentation hypotheses. Chapter 6 gives the full description of steps 5–6 and the word-level selection rule.

3.2 Group-wise bounding box normalization

Before polynomial features are generated, the raw ink coordinates need to be normalized to a common coordinate space. The position and scale of characters within a sample vary greatly; a large character (e.g., capital A) can extend many hundreds of device units, while a small character like a comma might span only a few dozen device units [13]. When coefficients are calculated directly from the raw character path data, the difference in size of the characters will dominate over their respective shapes, which is what matters most for classification purposes. This chapter describes the process by which the character samples in the database are normalized, and presents the rationale supporting this normalization process.

Motivation

Normalization is intended to obtain a representation for each type of character so that Legendre coefficients (see Chapter 4) remain unchanged under translation and isotropic scaling. Invariance to translation and scale is a standard requirement for shape features in the pattern recognition literature [53, 55]; rotation invariance is not enforced by the bounding-box normalization described below and is intentionally left out, since handwritten characters are read in a fixed orientation. For example, it would be beneficial if two instances of the same character (one written in a different location than another instance of the same character), regardless of the size used to write the second character by a different writer, could both correspond to the same class of features.

Each character group is normalized as a group, rather than individually by each stroke. It is this aspect which is critical for those cases where a single character can be formed from multiple strokes. For example, the letter i consists of a vertical body stroke, and a relatively small circular stroke above the body stroke. The circular stroke must be positioned approximately one stroke length directly above the body stroke for the resulting

coefficient vector to encode the spatial relationship between the two strokes. Otherwise, the body stroke would occupy all available space, leaving the circular stroke as a small normalized mark with little relation to the body stroke.

If normalization was performed at the individual stroke level, converting each stroke to have values ranging from $[0, 1]$, the body stroke would consume all available space while the circular stroke would be converted into an almost randomly sized shape and lose all connection to its companion body stroke. By normalizing at the group level, the bounding box encloses all strokes hypothesized to belong to the same character, and the relative positions of those strokes within the box are preserved. This covers two-stroke characters like *A* or *t*, three-stroke characters like # and many more.

Bounding box computation

For a character group $G_j^{(c)}$ containing one or more strokes, all sampled points from every stroke in the group are collected. The axis-aligned bounding box is then computed by taking the minimum and maximum coordinates across all of these points:

$$x_{\min} = \min\{x \mid (x, y) \text{ is a point in any stroke of } G_j^{(c)}\},$$

$$x_{\max} = \max\{x \mid (x, y) \text{ is a point in any stroke of } G_j^{(c)}\},$$

and similarly for $y_{\min}$ and $y_{\max}$. The bounding box width and height are then:

$$w = x_{\max} - x_{\min}, \qquad h = y_{\max} - y_{\min}.$$

All the strokes in this group share this bounding box. Each point in each stroke in the group is normalized with respect to $x_{\min}$, $y_{\min}$, and the scale factor. This frame of reference maintains the inter-stroke relationships such as offset, overlap, or crossing, as the relative positions of the stroke points are invariant in the normalized space. Figure 3.1 shows a multi-stroke character before and after the group-wise step.

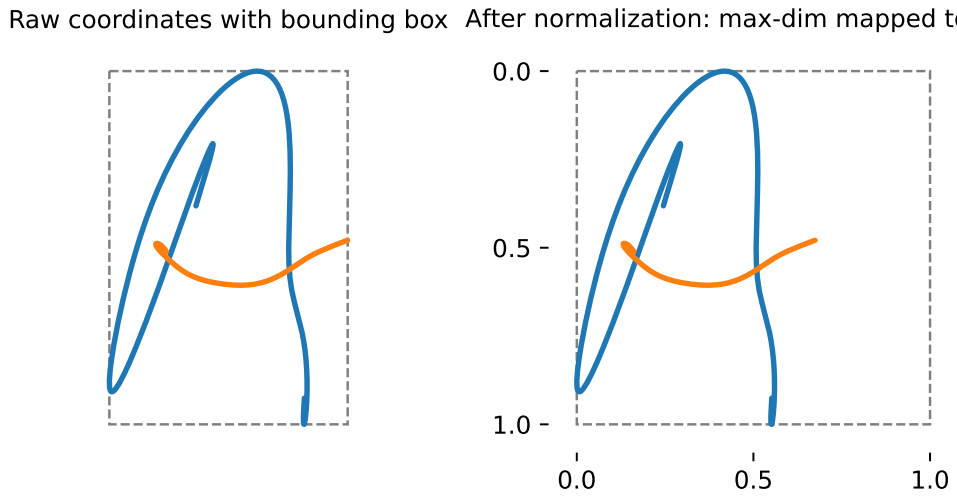

Figure 3.1: Group-wise bounding-box normalization. (a) Raw coordinates with the group bounding box. (b) After translation to the origin and division by $\max(w, h)$.

Aspect-preserving scale normalization

The scale factor is defined as the larger of the two bounding box dimensions:

$$\sigma = \max(w, h).$$

Each point (x, y) in the group is then mapped to normalized coordinates:

$$x' = \frac{x - x_{\min}}{\sigma}, \qquad y' = \frac{y - y_{\min}}{\sigma}.$$

The choice to divide by a single scale factor σ rather than independently by w and h is deliberate. Dividing by $\max(w, h)$ preserves the original aspect ratio of the character. Mazalov (2013) compares aspect-ratio-preserving normalization with height-only and integral-invariant normalizations and reports that aspect-ratio scaling is the suitable choice when rotational deformations dominate [6]. After normalization, the x and y coordinates are limited to the $[0, 1]$ interval along the longer axis of the character's bounding box. The shorter axis is then restricted to a subinterval of that interval and the length of that subinterval is proportional to the original aspect ratio of the character.

To illustrate why this matters, consider two simple examples: a tall narrow l and a wide m. For a tall, narrow l, the height is much greater than the width $h \gg w$, so $\sigma = h$. After normalization the y'-coordinates of the letter's strokes range over almost the whole $[0, 1]$ while the x'-coordinates occupy only the narrow subinterval $[0, w/h]$. For a wide m, the width is much greater than the height $w \gg h$, so $\sigma = w$. Now the x'-coordinates range over almost the whole $[0, 1]$ while the y'-coordinates occupy only the narrow subinterval $[0, h/w]$. If each axis of the coordinate system has independent limits of $[0, 1]$ then both letters fill the entire unit square, and all information regarding the aspect ratios of the different distributions of the x' and y' coordinates is lost. Therefore, the Legendre coefficients that could be obtained by examining a distorted version of the images would be unable to tell the difference between these visually distinct characters. Figure 3.2 illustrates the same letter under each of the two scaling rules.

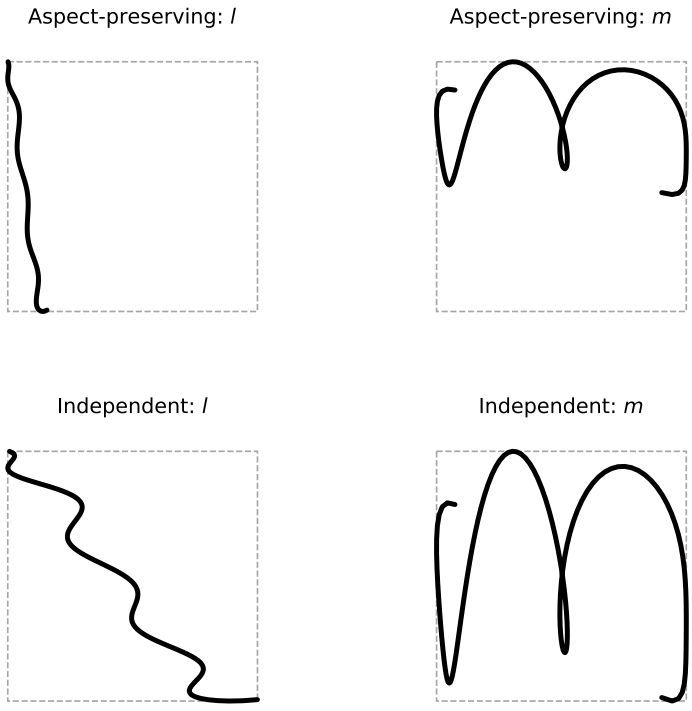

Figure 3.2: Aspect-preserving versus independent scaling on the letters l and m. Top row: aspect-preserving. Bottom row: independent x/y scaling.

More formally, the normalization removes exactly two degrees of freedom from the raw

coordinate data:

- **Translation**: by subtracting $x_{\min}$ and $y_{\min}$, the group is shifted so that its bounding box corner coincides with the origin.
- **Isotropic scale**: by dividing by σ, the group is scaled uniformly in both axes so that its largest dimension maps to $[0, 1]$.

Both are appropriate for handwriting recognition: the same character written at different positions or sizes on a page should get the same representation, while characters with different proportions should remain distinguishable.

Raw size retention

Although normalization removes the absolute scale from the coordinate representation, the raw bounding-box size

$$\sigma_{\text{raw}} = \max(w, h)$$

is recorded for each character group *before* normalization is applied. This value is needed because normalization can make geometrically dissimilar characters appear similar in coefficient space.

An example would be when small punctuation symbols are misidentified as larger characters with similar normalized shapes. The curvatures of a comma and a forward slash are relatively equivalent after normalization; each is a short curved/angled line. In the original ink, the comma is much smaller than the slash, typically a small fraction of the median character size in the word. Similarly, a period and a short vertical stroke (or a digit 1) can become difficult to distinguish once scale is removed. Mazalov and Watt (2012) observe that scale normalization makes very small characters, such as commas, periods, and dashes, shape-equivalent to larger characters with similar curvature, and propose size-aware classification schemes that adjust shape-based scoring using the pre-normalization size of the sample [12, 13]. The punctuation handling described in this section is inherited from that earlier work; the current UniPen evaluation uses a Latin-letters-and-digits database only, so the punctuation classes are not exercised in the experiments reported in Chapter 7. Figure 3.3 illustrates the problem.

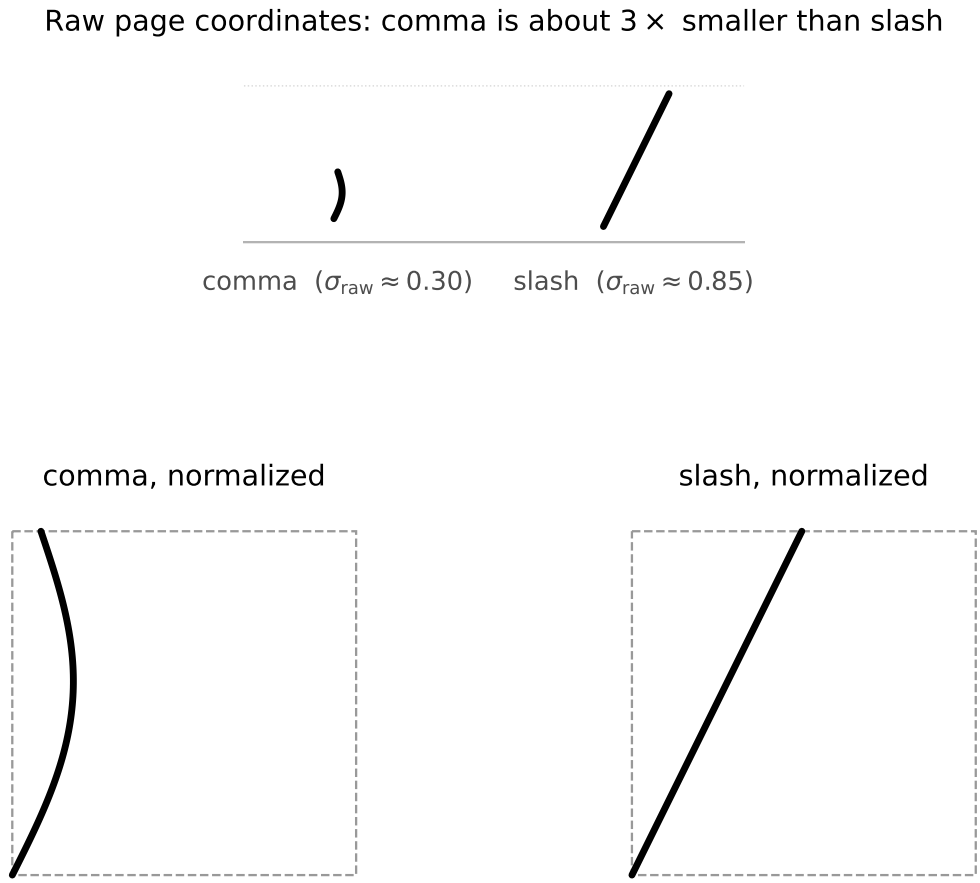

Figure 3.3: Why raw bounding-box size is retained. (a) A comma and a slash at their on-page sizes: the comma is roughly 4× smaller. (b) Both characters after group-wise normalization: the size signal is gone and the two strokes occupy the same unit box.

The fix is to retain the pre-normalization size and let the classifier flag the anomalies. The classes for small punctuation marks, namely the comma and period, are identified, and the length of the test character is divided by the average length of characters in that class. If the ratio is large (i.e. the test stroke is much longer than the typical stroke in the class), a penalty is applied to the classification distance to stop long strokes from being classified as a comma or period.

For letters, digits and other types of characters the raw size does not have any impact on the classification and so weights are all set to 1. This means that the sizes of the

characters are expected to change by a factor of a few and not be strongly penalized.

3.3 Arc-length parameterization

Each character group now sits in a common coordinate space after normalization by its bounding box. However, each stroke still requires an additional parameterization to allow for projection of the coordinate functions into the Legendre polynomial basis. Arc-length parameterization was used as the primary method for this purpose. Arc-length parameterization provides two advantages: first, it maps the coordinates of the strokes onto the interval $[-1, 1]$, which means that the Legendre coefficients from projecting the stroke coordinates will be independent of both the sampling rate and the writing speed.

Motivation

The original points of a stroke segment are indexed by the order that they are sampled: the first point sampled is p_1, the second is p_2, and so on. These indices do not correspond to any meaningful geometric property. A slow and precise writer will sample a curve with more points and smaller distances between them, while a fast and loose writer will sample the same curve with fewer points and larger distances between them. Thus, features derived from the index of the points will have high variance for the same object (geometric curve) drawn by different people [67, 1].

Zahn and Roskies (1972) parameterize a closed curve by arc length and expand the resulting intrinsic function as a Fourier series [54]. Applied to handwriting strokes, this gives a smooth parameterization with equal increments of the parameter for all cases. Two strokes with the same geometry but different sampling densities are then parameterized identically, and the coefficients derived from the Legendre expansion match. Golubitsky and Watt (2008) report that arc-length parameterization is essential to the curve and unaffected by writing-speed variation, and demonstrate Legendre fits computed from arc-length parameterized strokes [10]. An alternative line of work parameterizes pen trajectories by tangent-angle features instead of coordinate functions [68].

Cumulative arc length

Given a normalized stroke with n points $(x'_1, y'_1), (x'_2, y'_2), \ldots, (x'_n, y'_n)$, the cumulative arc length at point k is defined as:

$$\ell_k = \sum_{i=2}^{k} \sqrt{(x'_i - x'_{i-1})^2 + (y'_i - y'_{i-1})^2}, \qquad \ell_1 = 0.$$

The total arc length of the stroke is $L = \ell_n$. Each cumulative arc length is then normalized to the unit interval:

$$t_k = \frac{\ell_k}{L} \in [0, 1].$$

This produces a monotonically increasing sequence $t_1 = 0 < t_2 < \cdots < t_n = 1$, where each t_k represents the fraction of the total stroke length that has been traversed up to point k. Points that are close together on the curve (because the pen was moving slowly) receive similar t-values, while points that are far apart (because the pen was moving quickly) receive well-separated t-values.

Mapping to $[-1, 1]$

The normalized arc-length parameter $t \in [0, 1]$ is mapped to the interval $[-1, 1]$ via the transformation:

$$s_k = 2\,t_k - 1 \in [-1, 1].$$

This produces the parameter values $s_1 = -1,\ s_2, \ldots, s_{n-1},\ s_n = 1$, with the first point of the stroke at $s = -1$ and the last point at $s = 1$. The coordinate functions of the stroke are now defined at these parameter values:

$$x'(s_k) = x'_k, \qquad y'(s_k) = y'_k.$$

The interval $[-1, 1]$ is chosen to match the orthogonal polynomial basis used for feature extraction in Chapter 4.

The parameterization produces coordinate values $x'(s_k)$ and $y'(s_k)$ only at the discrete sample locations s_k. However, feature extraction using Legendre polynomials (Chapter 4) requires evaluating the coordinate functions at arbitrary values of $s \in [-1, 1]$. Since the

original data consists of discrete samples, the coordinate functions are reconstructed by piecewise linear interpolation between consecutive sample points [69].

For a query value $s \in [s_k, s_{k+1}]$, the interpolated coordinates are:

$$x'(s) = x'_k + \frac{s - s_k}{s_{k+1} - s_k}(x'_{k+1} - x'_k), \qquad y'(s) = y'_k + \frac{s - s_k}{s_{k+1} - s_k}(y'_{k+1} - y'_k).$$

At the boundaries, $x'(s) = x'_1$ for $s \leq -1$ and $x'(s) = x'_n$ for $s \geq 1$, and similarly for y'. This piecewise linear model assumes that the pen moves in approximately straight lines between consecutive samples, which is reasonable when the sampling rate is sufficiently high relative to the curvature of the stroke.

The result is a pair of continuous functions $x'(s)$ and $y'(s)$ defined over the full interval $[-1, 1]$, ready for the Legendre polynomial projection described in Chapter 4.

Properties of arc-length parameterization

Arc-length parameterization provides several properties that are desirable for handwriting recognition:

- **Speed invariance.** Two traces of the same geometric curve, one drawn slowly and one drawn quickly, receive the same parameterization because arc length depends only on the curve geometry and not on the rate at which it is traversed. This is important because writers vary their speed both within and across strokes.
- **Sampling invariance.** Two traces of the same curve sampled at different rates (e.g., by different capture devices) receive comparable parameterizations. Dense sampling produces many s_k values clustered along the curve, but the functions $x'(s)$ and $y'(s)$ remain the same because the underlying geometry has not changed.
- **Compatibility with Legendre polynomials.** The Legendre polynomials are orthogonal on the interval $[-1, 1]$ with respect to the uniform weight function. It is shown in the next chapter that arc-length parameterization leads to a uniform distribution of the curve in the parameter domain with respect to the geometric distance. This is exactly what is needed for uniform weighting of the Legendre inner product.

Therefore, the first few terms in the truncated series will correspond to the most significant, or geometric, components of the stroke.

Golubitsky and Watt (2008) report these properties in the context of online stroke modeling with truncated orthogonal series [10], and the choice of arc-length over time-based parameterization is shown there to be essential to the curve and unaffected by variations in writing speed.

3.4 Edge cases

The previous sections have assumed that every stroke from either a training or testing sample will be compatible with both normalization and parameterization. Unfortunately this does not always hold true for handwritten samples, therefore, this section covers all the different edge cases which have been identified in existing datasets and outlines how these are handled by the pipeline.

Zero-length strokes

A stroke may have near-zero arc length for several reasons:

- **Single-point strokes.** The user places the stylus on the writing surface and immediately lifts it. A sampling has occurred at exactly one point ($n = 1$) along this single stroke. There are two possible causes of such a case. First, the user may deliberately place a single dot (i.e., an i or a period). Secondly, the capturing device may record a very brief contact.
- **Coincident-point strokes.** The pen remains stationary or nearly stationary during a pen-down segment, producing multiple samples that are all within a small neighbourhood. The cumulative arc length L is positive but negligible (below the threshold 10^{-10}).

In both cases, the arc-length normalization $t_k = \ell_k / L$ is either undefined (when $L = 0$) or numerically unstable (when L is extremely small). The system handles this by switching

to a *uniform index-based parameterization*:

$$t_k = \frac{k-1}{n-1}, \qquad k = 1, \ldots, n,$$

for strokes with $n \geq 2$ points, and $t_1 = 0$ for single-point strokes. This produces an evenly spaced parameter sequence that allows the Legendre projection to proceed. The resulting coefficients reflect a nearly constant function, which is geometrically appropriate: a stroke with no spatial extent carries little shape information, and its coefficients should be close to zero for all non-constant basis functions.

Zero-area bounding boxes

A character group may have a bounding box with zero width, zero height, or both. This occurs when:

- All strokes in the group are single-point taps at the same location.
- All strokes in the group are perfectly horizontal (height zero) or perfectly vertical (width zero).
- The group contains only one stroke that is a single point.

When the scale factor $\sigma = \max(w, h)$ falls below 10^{-10}, it is clamped to $\sigma = 1$. The normalized coordinates then reduce to:

$$x' = x - x_{\min}, \qquad y' = y - y_{\min},$$

which are the raw coordinates shifted to the origin but unscaled. Since all points are nearly coincident in this case, the resulting values are close to zero, and the Legendre projection produces near-zero coefficients. This is a safe and consistent outcome: the character group adds almost no information to the classification, and the pipeline (KNN, convex hull ranking) assigns it a large distance to most class centroids. The low confidence that follows is the right answer for such inputs.

Multi-stroke groups with mixed quality

A character group can have regular strokes and special-case strokes. For example, for the letter *i*, there is a regular body stroke, and a small dot which is considered a single point tap. The bounding box and scale factor are calculated for the whole group and the scale is taken from the body stroke of the character. The small dot then has valid normalized coordinates (a small cluster centered near the top of the bounding box) and its arc-length parameterization is calculated as an index-based parameterization. The body stroke is parameterized as usual.

The above behavior is correct because the body stroke is used to normalize the stroke, thus positioning the small dot relative to that. The Legendre coefficients for the dot stroke will be similar to constants (because they represent a localized region in space), while the Legendre coefficients for the body stroke will accurately describe its true geometry.

Summary of fallback behaviour

Table 3.1 summarizes the edge cases and the corresponding fallback mechanisms.

Table 3.1: Edge cases in normalization and parameterization.

Condition	**Affected Stage**	**Fallback**
$\sigma < 10^{-10}$	Bounding box normalization	Clamp $\sigma = 1$
$L < 10^{-10}$, $n \geq 2$	Arc-length parameterization	Uniform index: $t_k = (k-1)/(n-1)$
$n = 1$	Arc-length parameterization	Single value: $t_1 = 0$

In all cases, the fallback produces a well-defined parameter sequence and a valid set of Legendre coefficients. No stroke is discarded or skipped at recognition time; every stroke in every character group contributes to the final coefficient vector, even if its contribution is minimal. The database-construction pipeline described in Chapter 5 applies a separate cleaning step that removes broken training samples (those with fewer than two coordinate points or zero bounding-box area), but this filter is applied only when building the reference database and never to incoming test traces.

3.5 Metadata

In addition to the normalized stroke coordinates, the normalization stage saves metadata for each of the character groups. This metadata is not used during the Legendre transform, but it is required by the classification process. These values are saved as part of the normalization, since they will be based on the normalized coordinate frame that has been determined individually for each group from the individual group-wise bounding box.

Stroke endpoints

For each stroke in a character group, the first and last points in normalized coordinates are recorded:

$$\mathbf{e}_i = \big((x'_{i1},\, y'_{i1}),\ (x'_{in_i},\, y'_{in_i})\big).$$

The coefficient vectors and endpoint pairs are then passed to the classification stage, where they are also used for the constraints based on adjacencies as shown in section 6.2. To classify a multi-stroke test character using an existing sample from a multi-stroke database, the classifier will need to generate all possible permutations of each test stroke relative to the sample strokes. Since the total number of possible permutations increases factorially with the number of strokes in the character, the cost grows quickly with stroke count and reduction of the permutation set is useful even when the absolute count is still small.

Multistroke recognition uses orthogonal series to combine successive strokes of a character into a single curve before generating the coefficients of the resulting curve; this allows the stroke sequence to be considered as part of the input for the classification [11]. In contrast, our pipeline has a reduction of possible permutations prior to combining successive strokes. Any permutation that contains strokes whose endpoint locations are farther away from each other than the predefined threshold will contain at least one stroke sequence that is likely not the original intended order of the strokes. The minimum distance of the four potential combinations of the consecutive endpoints of two adjacent strokes (start-to-start, start-to-end, end-to-start, and end-to-end) is calculated. Any permutation that includes a max distance greater than this predefined threshold will not be included in the subsequent search space. Since good handwriting typically produces coherent spatial arrangements of stroke orders after reducing the search space, it is highly improbable that the true stroke order would fall outside this reduced search space.

Additionally, instead of using the raw coordinate values of the endpoints, we use them in normalized form. This means that regardless of whether a particular group of characters has varying sizes, the distance threshold for determining if there exists an acceptable permutation of consecutive stroke endpoints will remain consistent.

Summary

This chapter explained how to transform raw digital ink into a format where features can be extracted. The first transformation is group-wise bounding-box normalization, which removes both location and scale; the second transformation is arc-length parameterization, which removes variance caused by variation in sample rate and stroke speed. Each stroke will now be defined as a series of continuous coordinate functions on $[-1, 1]$ along with some metadata. The next chapter describes how these normalized strokes are projected onto the Legendre polynomial basis to produce the compact coefficient vectors used for classification.

Chapter 4

Polynomial feature extraction

This chapter covers projecting the data described in Chapter 3 into a Legendre polynomial space, which will provide a means of distinguishing one shape from another based on the coefficients of the Legendre expansion. Each stroke comes in as a pair of continuous interpolated functions $x'(s)$ and $y'(s)$ on the $[-1, 1]$ interval (Section 3.3), and ends up with a pair of coefficient vectors $\mathbf{a}$ and $\mathbf{b}$ that describe its shape. A number of methods have been proposed for representing a curve as an expansion in terms of orthogonal basis functions such as Legendre and Chebyshev polynomials. Char and Watt (2007) used truncated polynomial expansions of Chebyshev polynomials to represent written symbols [4], and Golubitsky and Watt (2008) used Legendre expansions for their online stroke model [10]. These authors utilized both time and arc-length parameterizations.

As well as the work above, there have been many other works exploring similar ideas. For example, Hu (1962) showed that image moments were invariant under translation and rotation and thus could be used as invariant features to characterize shapes [53]. Similarly Zahn and Roskies (1972) presented a method using Fourier descriptors to represent closed curves along an arc-length parameterization [54], and they were later extended by Persoon and Fu (1977) to make them suitable for shape discrimination [70]. Finally, Teh and Chin (1988) studied families of orthogonal moment representations and found that Legendre and Zernike moments retained more compactly the shape information than did geometric moments [55], which motivated our use of an orthogonal polynomial basis here.

4.1 Legendre projection

Given a normalized stroke with interpolated coordinate functions $x'(s)$ and $y'(s)$ defined over the interval $[-1, 1]$, the projection uses the Legendre Basis $\{\phi_0, \phi_1, \ldots, \phi_d\}$ to determine the coefficients for these functions.

For each basis function ϕ_j, the x and y-coefficients are:

$$a_j = \langle x', \phi_j \rangle = \int_{-1}^{1} x'(s)\,\phi_j(s)\,ds, \qquad b_j = \langle y', \phi_j \rangle = \int_{-1}^{1} y'(s)\,\phi_j(s)\,ds.$$

These are expanded in the orthogonal Legendre basis, which has one basis function for each index ($j = 0, 1, \ldots, d$). Orthogonality means these modes are non-overlapping, so each pair (a_j, b_j) contributes a separate component of the stroke's geometry rather than mixing scales together.

- a_0 and b_0 encode the average position of the stroke along each axis (the constant term, or mean component).
- a_1 and b_1 encode the overall slope or direction of the stroke (linear term).
- Higher-order coefficients a_j, b_j for $j \geq 2$ encode finer geometric detail: smoother curves, inflection points, and oscillations.

Orthogonality means the coefficients are independent coordinates in the chosen basis: changing the value of one coefficient does not affect any other. Each coefficient contributes a unique piece of information about the shape. The truncated series $\sum_{j=0}^{d} a_j \phi_j(s)$ is also the degree-d polynomial that minimizes $\int_{-1}^{1} (x(s) - p(s))^2\,ds$ over all polynomials p of degree at most d; this is the standard $L^2[-1, 1]$ projection, discussed more thoroughly in Section 4.6. Figure 4.1 shows a single stroke together with its x- and y-coefficient vectors at degree 11.

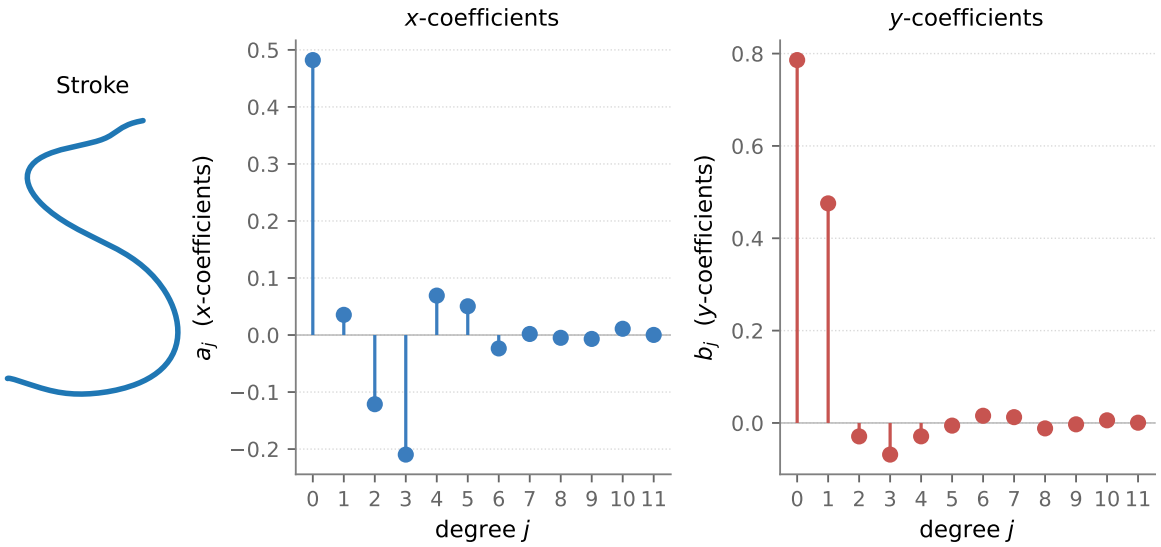

Figure 4.1: A single stroke and its degree-11 Legendre coefficient vectors. (a) Original stroke. (b) x-coefficients $a_0, \ldots, a_{11}$. (c) y-coefficients $b_0, \ldots, b_{11}$.

The degree d controls how much detail the representation retains. A higher degree captures more shape variation but also more noise; a lower degree smooths the representation but may lose discriminative features. The choice of degree used in this thesis is discussed in Section 4.5.

4.2 Basis construction

Before processing any strokes, the orthogonal Legendre basis has to be created. In this thesis, the basis is built by applying Gram–Schmidt orthogonalization (as described by Stoer and Bulirsch, 2002, in their treatment of orthogonal polynomial construction) [69] to the monomial basis $\{1, s, s^2, \ldots, s^d\}$ with respect to the $L^2[-1, 1]$ inner product:

$$\langle f, g \rangle = \int_{-1}^{1} f(s)\, g(s)\, ds.$$

The Gram-Schmidt process takes each monomial in order. For each one, it subtracts all components that lie along the previous basis functions, leaving a polynomial orthogonal to

those that came before. The first few basis functions are the standard Legendre polynomials $P_j(s)$:

$$\begin{aligned}\phi_0(s) &= 1,\\ \phi_1(s) &= s,\\ \phi_2(s) &= \tfrac{1}{2}(3s^2 - 1),\\ \phi_3(s) &= \tfrac{1}{2}(5s^3 - 3s).\end{aligned}$$

These are orthogonal on $[-1, 1]$ with $\langle \phi_i, \phi_j \rangle = (2/(2j+1))\,\delta_{ij}$. The pattern continues through ϕ_{11} for degree $d = 11$. Both this section and Section 4.6 use the standard orthogonal Legendre basis $\phi_j = P_j$, with the non-unit norms accounted for by the weights in the coefficient-space distance.

In the implementation, the Gram-Schmidt process is carried out symbolically on polynomial objects rather than numerically on sampled values. The inner products are evaluated exactly using symbolic integration on $[-1, 1]$. Because the basis is built exactly, no numerical error accumulates from approximate integration. The $d+1$ orthogonal polynomials are generated once and reused for each stroke's projection for the remainder of the recognition session.

The choice to build the basis with Gram-Schmidt rather than a closed-form recurrence like the Bonnet recursion for Legendre polynomials follows from the polynomial algebra library used in the implementation. Both methods produce the same basis, but the Gram-Schmidt construction has one advantage: it can be reused for other inner products, such as the Legendre-Sobolev inner product used by Alvandi and Watt (2020) in their package for handwriting recognition [8], without needing a separate derivation.

4.3 Numerical integration

The projection integrals $\int_{-1}^{1} x'(s)\,\phi_j(s)\,ds$ admit exact piecewise integration, since $x'(s)$ is a piecewise-linear interpolation through discrete sample points (Section 3.3) and the product of a polynomial with a piecewise-linear function can be integrated exactly on each

piece. The pipeline instead approximates them with adaptive numerical quadrature for simplicity and generality, splitting the integral into smaller subintervals and refining the estimate where the function is hard to approximate.

The integration method used is a custom adaptive five-point quadrature rule inspired by Newton–Cotes formulas, following the family described by Press et al. (2007) [71]. The interior nodes are placed at unequal positions inside each subinterval, so the rule is not standard Boole's rule. For a function $f(s)$ on an interval $[a, b]$, the algorithm proceeds as follows:

1. Evaluate f at the endpoints a and b plus three interior points that are not evenly spaced. The uneven spacing avoids resonance artifacts when the function has a regular pattern.
2. Compute a trapezoid estimate and a five-point Newton–Cotes-style estimate from these five values.
3. If the two estimates agree to within a set tolerance, return the five-point value.
4. Otherwise, split the interval into four smaller parts and repeat the same steps on each one. The allowed error on each subinterval is reduced to account for errors that accumulate over the recursion.

The method spends more evaluations on regions where the function changes quickly, like sharp turns, and fewer on smooth regions. The uneven spacing of interior points avoids resonance with periodic functions whose zeros fall on equally spaced grids, which would otherwise create false agreement between the two estimates.

For each stroke, the integration runs $2(d + 1)$ times: each of the $d + 1$ basis functions is applied twice, once to $x'(s)$ and once to $y'(s)$. At each evaluation point, $x'(s)$ or $y'(s)$ is obtained by linear interpolation between sample points. The basis function $\phi_j(s)$ is then evaluated as a polynomial. The product of these two values is the integrand, which is integrated over $[-1, 1]$ to produce one coefficient.

The default tolerance is 10^{-5}, well below the variation between handwriting samples. The source of error in the pipeline is the variability of human writing, not the integration step, so tighter tolerances would not improve recognition accuracy. Alvandi and Watt (2018) showed that real-time matrix formulations of this projection avoid the quadrature step entirely once the basis change is precomputed [72].

4.4 Multi-stroke coefficient representation

Many characters are drawn with more than one stroke. The letter i has a body and a dot (2 strokes), the letter t has a vertical stroke and a crossbar (2 strokes), and characters like A or f are often drawn with 2–3 strokes depending on the writer. The feature representation must account for this while still allowing characters with different stroke counts to be compared. Golubitsky and Watt (2009) addressed this for online multi-stroke symbols using orthogonal series [11].

Per-stroke projection

Each stroke in a character group is projected independently onto the Legendre basis, producing its own pair of coefficient vectors. For a character group with k strokes, the projection produces k pairs:

$$\big(\mathbf{a}^{(1)}, \mathbf{b}^{(1)}\big), \quad \big(\mathbf{a}^{(2)}, \mathbf{b}^{(2)}\big), \quad \ldots, \quad \big(\mathbf{a}^{(k)}, \mathbf{b}^{(k)}\big),$$

where each $\mathbf{a}^{(i)} = (a_0^{(i)}, \ldots, a_{11}^{(i)})$ and $\mathbf{b}^{(i)} = (b_0^{(i)}, \ldots, b_{11}^{(i)})$ are 12-element vectors (for degree $d = 11$). Each stroke independently preserves the per-stroke shape information: the coefficients for the dot of i reflect the geometry of the dot alone, not a blend of the dot and the body stroke, following the per-stroke representation adopted in Mazalov (2013) [6].

Flat vector layout

For classification, the per-stroke coefficient pairs are concatenated into a single flat vector. The layout alternates the x- and y-coefficients of each stroke in sequence:

$$\mathbf{v} = [\underbrace{a_0^{(1)}, \ldots, a_{11}^{(1)},\ b_0^{(1)}, \ldots, b_{11}^{(1)}}_{\text{24 values: stroke 1}},\ \underbrace{a_0^{(2)}, \ldots, a_{11}^{(2)},\ b_0^{(2)}, \ldots, b_{11}^{(2)}}_{\text{24 values: stroke 2}},\ \ldots]$$

The dimension of $\mathbf{v}$ depends on the stroke count:

Strokes	Dimension of v
1	24
2	48
3	72
4	96

This flat vector is the working representation for the rest of the pipeline: centroid computation, distance reduction, KNN search, and convex hull construction all act on it. Database samples are stored in the same format, so the weighted Euclidean distance between a test character and a database sample is a direct vector operation. Characters with different stroke counts are never compared: the classification pipeline first filters out classes whose stroke count does not match, following the stroke-count partitioning used by Golubitsky and Watt (2009) [11] and Mazalov (2013) [6] (Section 6.2).

Stroke ordering

Each stroke occupies a fixed slot in the flat vector (a block-concatenated dense vector) determined by its index: the first stroke uses the first 24 values, the second stroke uses the next 24, and so on. However, since multiple users may be drawing the exact same character but in a different order (one may start with the top-left leg while another starts with the bottom-right), the resulting flat vectors have matching coefficient blocks but in reversed positions. This creates an issue where if we directly compare these two strokes based solely upon their flat vector representations, we will see a very high distance due to simply having the stroke blocks in opposite orders, which does not help determine whether they represent the same character or not.

Therefore, the pipeline determines all possible stroke ordering combinations when determining whether two multi-stroke characters are equivalent. In addition, the flat vector representation was designed such that reversing any two sets of stroke blocks requires only O(1) time for changing a single reference point without requiring any copies of the associated data.

Consistency between database and pipeline

The flat vectors produced by both the offline database creation (Chapter 5) and the online recognition process have identical structure and formatting. Database centroid values were calculated as the average of each flat-vector value for all database samples. During the recognition phase, a distance was then computed between the flat-vector for the test character and each centroid value. If there existed some structural differences or differences in scaling coefficients between the two formats, that would result in a consistent error in all distances.

4.5 Degree and truncation

The degree d, set to 11 in this thesis, determines how many basis functions are used to approximate each coordinate function and how much detail the representation retains. Degree 11 produces 12 coefficients per coordinate per stroke. This section explains the reasoning behind the choice.

What each degree captures

The Legendre basis functions become more oscillatory as the degree goes up. Roughly:

- **Degrees 0–1** capture position and overall direction. A straight-line stroke is fully described by these two terms.
- **Degrees 2–4** capture curves, asymmetry, and overall shape. A simple curved line like a parenthesis or the letter c is well-represented at this level.
- **Degrees 5–8** capture finer structural features: inflection points, loops, and the proportions of different parts of the letter. This range matters for separating characters that share a broad shape but differ in detail, such as n versus m, or a versus o.
- **Degrees 9–11** capture the smallest details in writing: little hooks at the end of strokes, serifs, and small changes in curvature. Teh and Chin (1988) reported that

high-order orthogonal moments encode this fine detail with reduced redundancy compared to geometric moments [55]. These details carry little energy but help separate letters that look very similar.

Trade-off: accuracy versus robustness

Increasing the degree improves approximation accuracy: a degree-11 reconstruction follows the original stroke more closely than a degree-5 one. A higher degree also captures small uninformative variations such as hand jitter or device sampling noise. Teh and Chin (1988) characterized this noise sensitivity for orthogonal moments [55], and Mazalov (2013) observed the same effect for higher-degree Legendre-Sobolev coefficients of handwritten strokes [6]. Past a point, the reconstruction fits the noise rather than the underlying shape.

Truncation at degree d acts like a low-pass filter: it keeps the geometric features that are consistent across instances of a character and discards high-frequency variation that is not useful. Zahn and Roskies (1972) made this argument for Fourier descriptors, noting that higher-order terms primarily carry noise [54]. Golubitsky and Watt (2008) reported that truncated Legendre series approximate handwritten coordinate functions well at moderate degrees and that higher-degree coefficients become numerically unreliable [10]. The truncated representation is more reliable for distance-based comparison because it leaves the noise out of the vector. Figure 4.2 shows the same stroke reconstructed at three truncation degrees.

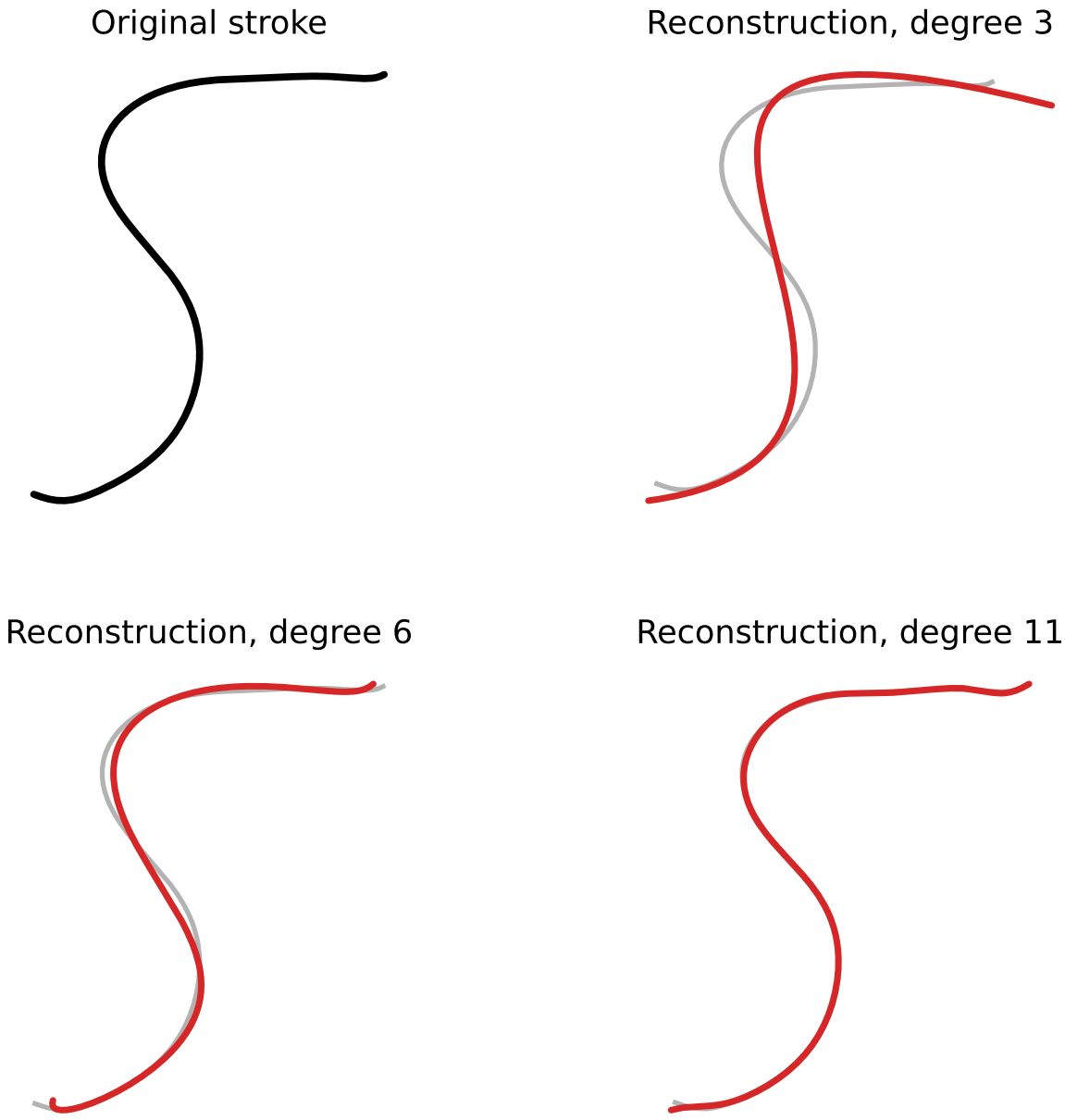

Figure 4.2: Stroke reconstruction at increasing truncation degree. (a) Original stroke. (b) Degree 3. (c) Degree 6. (d) Degree 11.

Why degree 11

The choice of degree 11 follows prior work in the orthogonal-series approach to handwriting recognition. Char and Watt (2007) used truncated polynomial series at degrees 3, 6, and 10 to represent handwritten mathematical symbols [4]. Golubitsky and Watt (2009) adopted truncation order $d = 12$ for Legendre-Sobolev expansions and reported that truncation orders between 10 and 15 give curves visually indistinguishable from the originals [11], and Mazalov (2013) reported $d = 12$ as the optimum recognition truncation on his symbol collection [6]. In earlier iterations of the pipeline used in this thesis, degree 10 was used (producing 11 coefficients). The change to degree 11 was motivated by two observations:

1. Characters like r and n, or u and v, differ only in small shape details, and the extra coefficient adds room for the curvature variations that separate such letters.
2. The cost of the extra coefficient is minimal: each stroke's vector gains 2 values, one for x and one for y, so a single-stroke character goes from 22 to 24 dimensions. This change has almost no effect on distance computation time or storage.

Going beyond degree 11 did not improve recognition accuracy and raised the risk of overfitting to random noise. This matches the analysis of Mazalov (2013), who reported that approximation error stops decreasing past this range and that higher-degree coefficients become numerically unreliable [6].

Consistency

The pipeline fixes degree 11 everywhere. Database samples are projected to degree 11 when the database is built, class centroids are averaged from degree-11 vectors, and at recognition time test characters are projected to degree 11 to match. This ensures all distances are computed in the same feature space with matching dimensionality.

4.6 L^2 curve distance in coefficient space

One of the main benefits of orthogonal polynomials is that distances between coefficient vectors have a direct interpretation: they measure how far apart the reconstructed curves are. Golubitsky and Watt (2010) established this connection for distance-based classification of

handwritten symbols [7]. When the coordinate functions are expanded in an orthogonal polynomial basis, the norm of the difference of two truncated curves is a weighted Euclidean norm of the coefficient differences, with diagonal weights equal to the squared norms of the basis functions. For the standard Legendre basis under the L^2 inner product these factors are $2/(2j+1)$; for a normalized basis they would be absorbed into the coefficients. This section derives the relationship and explains its role in the classification pipeline.

From curves to coefficients

Consider two coordinate functions approximated in the orthogonal Legendre basis:

$$f(s) \approx \sum_{j=0}^{d} \alpha_j \phi_j(s), \qquad g(s) \approx \sum_{j=0}^{d} \beta_j \phi_j(s).$$

The L^2 distance between these two functions on $[-1, 1]$ is:

$$\|f - g\|_{L^2}^2 = \int_{-1}^{1} \big(f(s) - g(s)\big)^2 \, ds = \int_{-1}^{1} \Big(\sum_{j=0}^{d} (\alpha_j - \beta_j)\, \phi_j(s)\Big)^2 \, ds.$$

Since the Legendre basis is orthogonal with $\langle \phi_i, \phi_j \rangle = (2/(2j+1))\, \delta_{ij}$, the cross terms vanish and each diagonal term carries $\|\phi_j\|_{L^2}^2 = 2/(2j+1)$ as a weight. This gives the weighted Euclidean distance in coefficient space:

$$\|f - g\|_{L^2}^2 = \sum_{j=0}^{d} \frac{2}{2j+1} \, (\alpha_j - \beta_j)^2.$$

This is Parseval's identity applied to the truncated unnormalized-Legendre expansion: the L^2 distance between two curves equals the weighted Euclidean distance between their coefficient vectors, with weights $w_j = 2/(2j+1) = \|\phi_j\|_{L^2}^2$ correcting for the non-unit norms of the orthogonal Legendre polynomials. Persoon and Fu (1977) used Euclidean distance in Fourier-descriptor space for shape discrimination [70], and Golubitsky and Watt (2010) applied a related distance on Legendre-Sobolev coefficient vectors to classify handwritten symbols [7]. Curves that look similar (small L^2 distance) produce coefficient vectors that are close together under this weighted metric, and vice versa.

The weights and what they encode

The weights $w_j = 2/(2j+1)$ are the norm-correction factors for unnormalized Legendre coefficients. They decrease with degree:

j	0	1	2	5	8	11
w_j	2.000	0.667	0.400	0.182	0.118	0.087

Low-order coefficients (which capture overall shape) contribute the most to the distance, while high-order coefficients (which capture fine detail and noise) contribute the least. Zahn and Roskies (1972) made the same observation for Fourier descriptors, where lower-order terms carry the dominant shape content and higher-order terms primarily encode noise [54]. This is geometrically appropriate: two strokes that differ in their broad curves (low-order) are visually more different than two strokes that differ only in a small hook at the endpoint (high-order).

Application in the pipeline

The L^2 weights $w_j = \frac{2}{(2j+1)}$ are used at the class-formation stage to compute each class radius from the spread of its members (Chapter 5). All recognition-time distances (centroid-based class reduction, KNN (Section 6.2), and convex hull creation (Section 6.3)) use the Sobolev weights described in the next subsection. When dealing with multiple strokes for a character, the weights will be applied on a stroke basis. The same weight vector will be applied to the 12 x coefficients and 12 y coefficients, and then the total distance will be the square root of the sum over all of the strokes.

Therefore, every distance comparison made within this system can be interpreted directly. It is measuring how different two characters *appear visually* as curves, not simply how close together their coefficient vector representations happen to be in some abstract space. This relationship between coefficient-space distance and visual similarity is a key property of the orthogonal series representation that provides much of the foundation for this choice of approach, established by Persoon and Fu (1977) for Fourier Descriptors [70] and extended by Golubitsky and Watt (2010) for handwritten symbol classification [7]. Therefore, this is one of the primary reasons that this methodology was chosen for this thesis.

Comparison with Sobolev weights

Another type of weighting scheme is the Sobolev-style degree weights

$$w_j = 1 + \alpha\, j^2,$$

with $\alpha = 0.3$ in this thesis. Mazalov and Watt (2012) described how a Legendre-Sobolev inner product applied to handwritten character recognition yielded better results than an unweighted L^2 product for both isolated and in-context recognition [12]. Alvandi and Watt (2020) then created the Legendre-Sobolev package based on this same inner product [8]. These weights increase as the order increases, which makes it possible to assign more importance to the middle and higher order coefficients that provide information about the curvature and finer details of the shape. In this thesis the basis itself is the ordinary orthogonal Legendre basis (not the Legendre-Sobolev basis), so the coefficients are unnormalized Legendre coefficients; the Sobolev weights are applied only to the distance metric. The proposed pipeline employs the two weighting schemes at different stages of the pipeline. The L^2 curve-distance weights $w_j = 2/(2j+1)$ are used for forming classes and computing class radii (Chapter 5), whereas the Sobolev weights ($\alpha = 0.3$) are used for computing all distances involved in the classification process (centroid reduction, KNN, and hull ranking) (Chapter 6). The use of the L^2 scheme allows for maintaining proportionality among class index-distances and curve-distances, and the use of the Sobolev scheme allows for separating characters that have identical positions but exhibit differences in terms of their curvatures. The weighted Euclidean norm in each metric is itself well-defined. What is not rigorous is the mixing of the two schemes (computing radii under L^2 weights while computing test-time distances under Sobolev weights), since the radius and the distance it bounds or normalizes are then measured in different metrics; this mixing is therefore treated as a heuristic, and the pruning and confidence formulas in Chapter 6 that rely on the class radius should be read in that light.

Summary

The polynomial feature extraction phase converts normalized stroke sequences into coefficient vectors based upon the representation of each stroke as a function of an eleven-degree orthogonal Legendre basis. Numerical integration was applied to produce twelve coeffi-

cients per coordinate. In addition to representing single-stroke characters, multi-stroke characters are represented by the concatenation of the coefficients produced for individual strokes into one-dimensional vectors. Two weighting schemes operate over the same coefficient space: the L^2 curve-distance weights $w_j = 2/(2j+1)$, used for forming classes and computing class radii, and the Sobolev weights $w_j = 1 + \alpha j^2$ with $\alpha = 0.3$, used in the classification pipeline. All of this information (the aforementioned coefficient vectors, along with additional metadata from Chapter 3) will be combined to form the total data presented to the classification phases outlined in subsequent chapters.

Chapter 5

Class database construction and indexing

5.1 Overview

In order for the recognition pipeline discussed later in the chapters to be able to look up unknown input, there must be a reference database of labeled handwriting examples to compare it against. This chapter explains how this database was constructed, organized and indexed so that the search for matching features at recognition time can occur quickly. The idea of first using a coarser classifier to reduce the number of candidates prior to further refinement in which remaining candidates are compared in detail comes from Golubitsky and Watt (2010) [7] who utilized a Manhattan-distance-based pre-classifier as their *coarse* classifier followed by a convex hull stage as their *fine* classifier in their distance-based classification research.

The per-class summary statistics which include an average (or center) and a ball formed around that center where the ball has its radius defined by a percentile, come from Mazalov (2013) [6] who used these summaries to characterize the range within which most instances of a particular class fall.

The first step in creating the reference database is to collect raw handwriting data from multiple sources and store it in InkML format [2]. Each entry in this collection represents one or more individual pen strokes that collectively make up a single character. After normalizing the raw strokes and applying the Legendre projection method as described in Chapters 3 and 4, we obtain a fixed-length feature vector for each stroke. If an example contains more than one stroke, the individual feature vectors representing those strokes are combined into a single vector.

In addition to stroke sequence, the second level of grouping is based on the similarity of the coefficient vectors for the samples. In essence, when we have samples with the

same stroke count and character label, these samples can be grouped together into *classes* because of similarities of their respective coefficient vectors. This classification process follows what Duda and Hart (1973) referred to as the standard classification problem in Pattern Classification [73]. Each letter will typically result in multiple classes that capture different aspects of writing such as an allograph or handwriting style. Schomaker and Teulings (1991) [74], Aksela (2007) [75], and Bahlmann and Burkhardt (2004) [22] have used allographs as the most basic units of classification for Prototype-Based Recognition. For example, the letter A, written using two strokes, would likely have at least two classes representing instances of A where the crossbar is oriented from left to right and those where it is oriented from right to left. Mazalov (2013) has also documented splitting labels in the same manner [6].

The data is stored in two separate files. The first file contains information about each sample, including its coefficient vector and which class it belongs to. The second file is a class-level indexing file containing a pre-computed centroid and radius for each class. When doing recognition, the class-level index allows the system to narrow down a large number of possible classes to a smaller group of candidates before having to do more costly comparisons with the sample-level information as described in section 6.2. The use of inexpensive initial classifiers to reduce candidate sets prior to doing more expensive comparisons is seen in Golubitsky and Watt (2010) [7] and in the prototype set pruning technique developed by Vuori et al. (2002) [23].

5.2 Data sources

The database draws on three sources of online handwriting data.

UniPen. UniPen is an open-access collection of online handwriting captured from a wide range of writers, devices, and input conditions, introduced by Guyon et al. (1994) [62]. The version used here is the train_r01_v07 release archived on Zenodo [9]. The full collection contains isolated characters, words, and sentences; only the isolated characters are used in this work. Each sample records the pen trajectory as a sequence of (x, y) coordinates with associated timestamps. UniPen provides 58,333 samples, which is 76% of the final database.

CROHME. CROHME (the Competition on Recognition of Online Handwritten Mathematical Expressions) distributes InkML files of handwritten mathematical expressions with stroke-level symbol annotations. The latest competition description comes from Mahdavi et al. (2019) [40], and Mouchère et al. (2016) give a retrospective survey of the earlier editions [39]. The annotations attach each pen stroke to a specific symbol in the expression, so individual Latin characters can be pulled out of multi-symbol formulas. After extraction, CROHME contributes 8,272 characters, around 11% of the database. Since mathematical expressions lean on uppercase letters and on certain lowercase letters such as x, y, n, and k, this data broadens the stylistic range for those labels.

Author-collected samples. An extra 1,355 samples were collected from five writers using tablet devices. Each writer produced samples for all 62 character labels. These samples cover styles that the two public collections do not include.

Combined total. The remaining 8,468 samples (11%) come from earlier iterations of the extraction pipeline applied to the same CROHME and UniPen sources. These went through a separate normalization path and were kept after they validated cleanly against the rest of the database. These samples were re-extracted from the same source UniPen and CROHME InkML files through an earlier version of the normalization code, and duplicates against current-source samples are avoided by `sample_id` matching, where each `sample_id` is source-derived as a hash of the source filename together with the symbol index within that file, so the same InkML symbol produces the same identifier regardless of which extraction pass produced it. Table 5.1 gives the source distribution.

Source	Samples	Share
UniPen	58,333	76.3%
CROHME	8,272	10.8%
Earlier extraction pipeline	8,468	11.1%
Author-collected	1,355	1.8%
Total	**76,428**	**100%**

Table 5.1: Distribution of samples across data sources.

5.3 Filtering to Latin characters and digits

The combined raw data contains Latin letters, digits, Greek letters, mathematical operators, brackets, arrows, and multi-character function names such as sin and lim. The recognition pipeline targets Latin handwriting and numerals, so the database keeps only the 52 Latin letters (A–Z, a–z) and 10 digits (0–9). Everything else is dropped.

The same filter also removes samples with broken stroke data: those with fewer than two coordinate points per stroke, those with zero bounding-box area, and those whose InkML markup could not be parsed. A similar verification pass was applied by Vuurpijl et al. (2004) to the public UniPen devset to clean up mislabeled and badly segmented samples [63]. Broken samples are removed from the reference database, since the database is built offline and there is no harm in dropping bad rows. Test-time input is treated differently. The pipeline never throws a broken stroke away; it runs the stroke through the same projection and lets the resulting low confidence speak for itself. After filtering, 76,428 samples remain across 62 labels.

5.4 Normalization and coefficient extraction

Each sample runs through the normalization and projection pipeline described in Chapters 3 and 4. The steps are summarized again here because the database construction adds one extra stage on top: direction canonicalization.

Bounding-box normalization. All strokes of a character go inside a single bounding box. The strokes are shifted so the box origin sits at $(0, 0)$, then scaled by $\max(\text{width}, \text{height})$ so the longer dimension maps to $[0, 1]$. The aspect ratio is preserved.

Per-stroke projection. Each stroke is reparameterized by cumulative arc length on $[-1, 1]$ and projected onto the orthogonal Legendre basis up to degree $d = 11$. This gives 12 coefficients for the x-component and 12 for the y-component (Section 4.1), a 24-dimensional vector per stroke.

Direction canonicalization. Two writers may draw the same stroke in opposite directions. A vertical stroke of "l" drawn top-to-bottom and one drawn bottom-to-top are geometrically identical but produce different coefficient vectors [76]. To get rid of this ambiguity, each stroke is assigned a canonical direction using a geometric heuristic based on the displacement $(\Delta x, \Delta y)$ from start to end:

- **Vertical-dominant** ($|\Delta y| > 1.1 \cdot |\Delta x|$): the canonical direction is top-to-bottom. If the stroke was drawn upward ($\Delta y < 0$), it is reversed.
- **Horizontal-dominant** ($|\Delta x| > 1.1 \cdot |\Delta y|$): the canonical direction is left-to-right. If the stroke was drawn right-to-left ($\Delta x < 0$), it is reversed.
- **Neither dominant**: the signed area enclosed by the stroke path is computed using the shoelace formula. A negative signed area (counter-clockwise winding) triggers reversal to enforce clockwise winding.

Reversing a stroke in coefficient space negates the odd-degree coefficients for both x and y. This follows from the parity of the Legendre polynomials: $\phi_j(-s) = (-1)^j \phi_j(s)$. Even-degree coefficients stay the same, and odd-degree coefficients flip sign.

This is applied to both database samples and test input. The same heuristic with the same thresholds runs against database samples and test data, so any canonicalization the database depends on is reproduced exactly at recognition time. It removes direction as a source of variation and also removes the need to search over 2^n direction combinations at match time. Without a fixed direction, each stroke can be drawn in either of two orientations, and a multi-stroke recognizer such as that of Golubitsky and Watt (2009) [11] carries that ambiguity through the joined coefficient vector unless the directions are resolved beforehand. The heuristic fails on rotated or mirrored handwriting, where the dominant-axis check assigns a direction that is geometrically opposite to the writer's intended stroke.

Multi-stroke concatenation. For a character with n strokes, the per-stroke vectors are joined into a single vector of dimension $24n$. Golubitsky and Watt (2009) used the same per-stroke join when they extended Legendre-Sobolev classification to multi-stroke symbols [11].

5.5 Class formation

The normalized database contains tens of thousands of coefficient vectors, where each vector has been labeled with a character label and a stroke count. Next we need to divide these vectors into *classes*: sets of samples which all share the same label, the same stroke count, and whose shapes in coefficient space are similar.

Many letters require multiple classes. For instance, the letter a is often written either as one continuous stroke or as an arc with a downward stroke. Thus, there are two distinct versions of this letter form based upon their stroke count. Within the single stroke version, a print version of a and a script version of a will have very dissimilar coefficients in coefficient space and therefore belong to distinct class families. These distinctions match the allographic-prototype model structure Aksela (2007) used in her thesis on Adaptive Online Recognition [75].

Two-phase classification. The first phase begins with a small subset of the data that has been manually classified. This subset contains *seed* classes for which a centroid will be computed using weighted coefficients. Then all samples that have the same label as the seed class and the same number of strokes will be assigned to the closest seed centroid. The overall strategy to create separate allographic groups based on clustering an entire set of samples labeled as being part of the same class was also done by Bahlmann and Burkhardt (2004) [22], however, they used agglomerative hierarchical grouping for clustering and this process uses nearest centroid assignment. The goal is to keep all of the manual classifications intact and pull in many thousands of samples that were left out of the manual classification.

The second phase includes any combination of labels and stroke counts that were included in the seed classes. To determine how these should be grouped, k-means clustering is performed using the K-means algorithm developed by MacQueen (1967) [77] and Lloyd (1982) [78] and further adapted for use in handwriting prototypes by Aksela (2007) [75]. In order to determine the number of clusters (k), we start with $k = 1$ and incrementally increase the value of k until the largest distance from any point in a given cluster to its centroid is less than some pre-defined threshold; alternatively we can limit our search to values of $k < 8$.

Naming. Each class gets an identifier that encodes its label, its stroke count, and a sequential index. The class `A_2strokes_class1` is the first class of two-stroke A samples. This makes it easy to filter by label or by stroke count at recognition time.

The procedure produces 3,237 classes across the 62 labels, with an average of 24 samples per class. The distribution is uneven. Common lowercase letters like e (4,311 samples) have over 60 classes, while rare uppercase letters like Q (396 samples) have fewer than 20. Figure 5.1 shows several distinct allographs of the letter a taken from the database.

Class-formation parameters. Table 5.2 lists the exact values used during class formation. The procedure is deterministic: the same input data with the same parameter values produces the same set of classes on every run.

Parameter	**Value**
Seed set size (samples)	1,626
Seed set size (classes)	709
Distance threshold τ	0.5
Maximum k	8
KMeans `n_init`	10
KMeans `max_iter`	300
KMeans `tol`	10^{-4}
KMeans `random_state`	42
Determinism	deterministic

Table 5.2: Class-formation parameters used in the two-phase classification procedure.

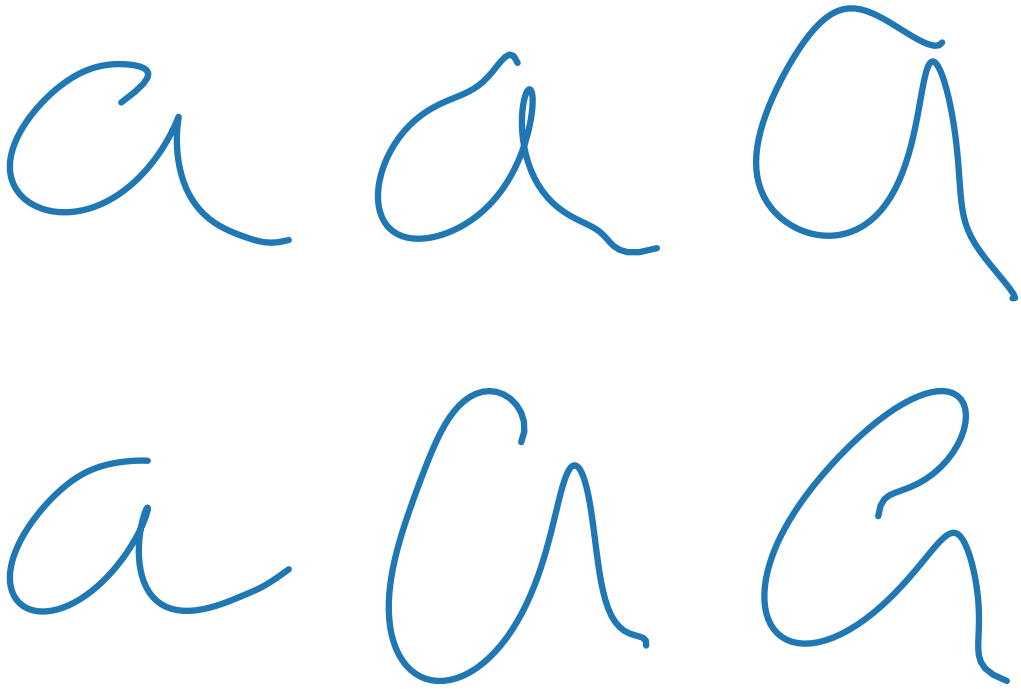

Figure 5.1: Eight allographs of the letter a from the reference database, reconstructed from their stored Legendre coefficient vectors.

5.6 Data cleaning and augmentation

Cleaning. After the classes were formed, each one was inspected by rendering its samples from their stored coefficient vectors. Any sample that did not look like a natural human version of its labeled character was removed. The cleaning was done by the author. Each class was rendered, visually inspected, and bad members marked for removal. Two rounds of review went through the database, targeting three kinds of problem samples:

- Corrupted or incomplete stroke data that reconstructed to a meaningless shape instead of a readable letter.
- Mislabeled extractions, where a stroke from a multi-symbol expression had been attached to the wrong character during the extraction step. Bahlmann and Burkhardt (2004) report the same problem in the UniPen database and strip small clusters of outliers for the same reason [22].

- Samples that were technically readable but written in a style too far from normal handwriting to help with classification. Keeping these around pulled accuracy down on the rest of the database.

Figure 5.2 shows four examples that were flagged for removal during this pass.

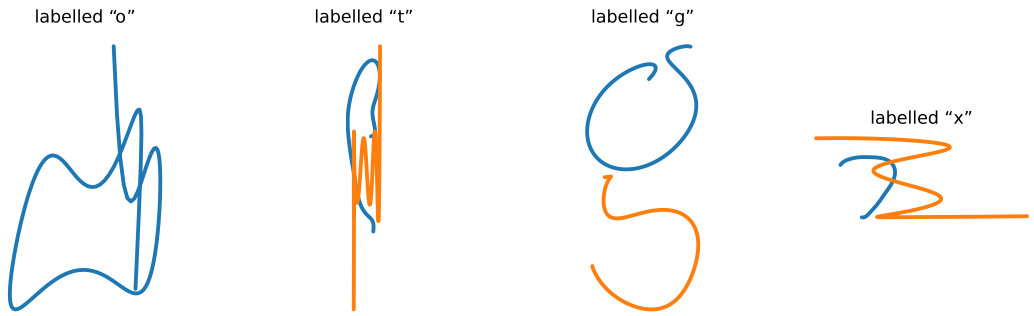

Figure 5.2: Four samples flagged for removal during cleaning, reconstructed from their stored coefficients before deletion. Above each panel: the label the sample was originally tagged with. The shapes do not match those labels: the `o` panel is a chaotic loop, the `t` panel is a vertical stem with a zigzag instead of a crossbar, the `g` panel has the bowl and the descender drawn out of place, and the `x` panel reads more like a horizontal zigzag than two crossing diagonals.

Augmentation. After cleaning, the uppercase letters were clearly underrepresented. The letter *E*, for example, had only 499 samples against 4,311 for lowercase *e*. To fix this, more uppercase samples were pulled in from a wider variant of the database that holds around 128,000 samples across a larger symbol set. Each candidate was run through the existing database in a leave-one-out test, and any sample that came back misclassified was thrown out as noisy. This is the misclassification-based filter that Wilson and Martinez (2000) proposed in their survey of instance-reduction techniques for nearest-neighbour learning [60], combined with the outlier-removal practice Bahlmann and Burkhardt (2004) describe for UniPen training data [22]. The pass added 4,179 verified samples, concentrated in the uppercase letters A through Z.

Class centroids and radii were then recomputed on the final dataset.

5.7 Storage format

The database is split into two files. One file holds an individual record of each sample and the second holds an already computed class index. Dividing the data in this manner allows the recognition engine to load the small class index when it starts up and reduce its search area. Then after reducing the number of possible classes at lower computational cost than computing all distances, the engine can access the sample records only for the reduced set of candidates. The concept of performing an initial inexpensive candidate reduction process prior to full distance computation also has been explored by Golubitsky and Watt (2010) [7], as well as in the prototype-set architecture developed by Vuori et al. (2002) [23].

5.7.1 Sample records

Each sample occupies one line of a JSON-lines file. The fields are:

- `sample_id`: a unique identifier derived from the source dataset and a hash of the stroke data.
- `class_id`: the assigned class (e.g. `A_2strokes_class1`).
- `char_label`: the character this sample represents.
- `stroke_count`: the number of strokes.
- `degree`: the polynomial degree (11).
- `coeffs`: an array of per-stroke coefficient objects, each with an `x` and a `y` array of 12 Legendre coefficients.
- `norm_bbox`: the bounding box of the character before normalization. The width and height are retained for size-based weighting during classification [13].

The coefficients are stored already in canonical direction, so distance computation can use them straight from disk without any further transformation. A sample with n strokes ends up with a flattened vector of dimension $24n$.

5.7.2 Class index

The class index is a single JSON file with one record per class:

- `class_id`, `char_label`, `stroke_count`, `dim`: identifying fields.

- `centroid`: the mean coefficient vector of all members.
- `radius`: a distance threshold chosen so that 90% of class members fall within it. This gives a tighter estimate of class spread than the maximum distance, which can be skewed by a single outlier.
- `max_radius`: the maximum distance, stored as an upper bound.
- `n_samples`: the member count.
- `avg_raw_size`: the average pre-normalization bounding-box size, used for size-based weighting [13, 6].

The index has 3,237 entries against 76,428 sample records. At recognition time, the index narrows the search space before any per-sample distance computation gets started.

5.8 Centroid and radius computation

Each class is summarized by a centroid and a radius. These two quantities drive the class reduction step described in Chapter 6.

Centroid. The centroid of class k is the arithmetic mean of its member vectors:

$$\mathbf{c}_k = \frac{1}{|C_k|} \sum_{\mathbf{s} \in C_k} \mathbf{s}.$$

Distance weights. Distances in coefficient space use a weighted Euclidean metric. Two weighting schemes show up in this work.

The *L2 curve-distance weights* $w_j = 2/(2j+1)$ make the coefficient-space distance equal to the L^2 distance between the reconstructed curves [8]:

$$\|f - g\|^2_{L^2[-1,1]} = \sum_{j=0}^{d} \frac{2}{2j+1} (c_j^f - c_j^g)^2.$$

Low-order coefficients (overall shape) dominate, and high-order coefficients (fine detail) get less weight. This scheme is used for class formation and centroid computation.

The *Sobolev weights* $w_j = 1 + \alpha j^2$ increase with degree. Mid and high-order coefficients carry more weight under this scheme. With $\alpha = 0.3$, these weights help separate characters that sit at similar positions but differ in curvature, such as the pairs n/m, h/k, and E/k. The Sobolev scheme is used in the classification pipeline (Section 6.2).

Radius. The radius of class k is the distance from the centroid that encloses 90% of the class members. Mazalov (2013) used the same percentile-radius definition when characterising class spread in his thesis, with a 75% threshold instead of 90% [6]. Formally, let $d_i = \|\mathbf{w} \odot (\mathbf{s}_i - \mathbf{c}_k)\|_2$ be the weighted distance from sample $\mathbf{s}_i$ to the centroid. The distances $\{d_i\}$ are sorted, and the radius is the value at the 90th position in that sorted list:

$$r_k = d_{\lceil 0.9 \cdot |C_k| \rceil}.$$

So 90% of samples sit inside the ball of radius r_k, and the remaining 10% (the likely outliers) fall outside it. Using this threshold instead of the maximum distance stops a single unusual sample from inflating the class size. For single-sample classes, the radius is set to a small positive constant to avoid division-by-zero errors later on.

The centroid and radius together define a ball in weighted coefficient space that roughly encloses the class. At recognition time, this ball supplies a pruning heuristic: classes whose centroid-plus-radius cutoff sits far from the test vector are skipped before any sample-level comparison. Because the radius is computed under the L^2 curve-distance weights while the recognition step uses Sobolev weights, the cutoff is not a guaranteed mathematical bound on the distance to class members, but it works well in practice. Chapter 6 describes how that cutoff drives the class reduction step.

5.9 Database summary

Table 5.3 summarizes the final database.

Property	Value
Character labels	62 (A–Z, a–z, 0–9)
Total samples	76,428
Total classes	3,237
Average samples per class	24
Polynomial degree	11 (12 coefficients per axis)
Coefficients per stroke	24
Stroke counts represented	1, 2, 3, 4
1-stroke samples	52,873 (69%)
2-stroke samples	20,707 (27%)
3- and 4-stroke samples	2,848 (4%)

Table 5.3: Summary of the final character database.

Chapter 6

Recognition pipeline

This chapter gives the full description of the recognition algorithm; other chapters refer back to it. The pipeline receives a handwritten word as input (a list of pen strokes) and returns a predicted word. The recognition pipeline has three steps. The first step performs trace grouping, determining what strokes form the same letter. In the second stage k-nearest neighbour classification is performed; the coefficient vector of each letter group is compared to the reference database to find a label for that letter group. In the third stage convex hull ranking is performed, assigning geometric confidence to each predicted letter. The structure of the pipeline is based on the functional-coefficient approach established by Char and Watt [4] and expanded further by Mazalov [6]. A full block diagram of the recognition pipeline can be found in Fig. 6.1.

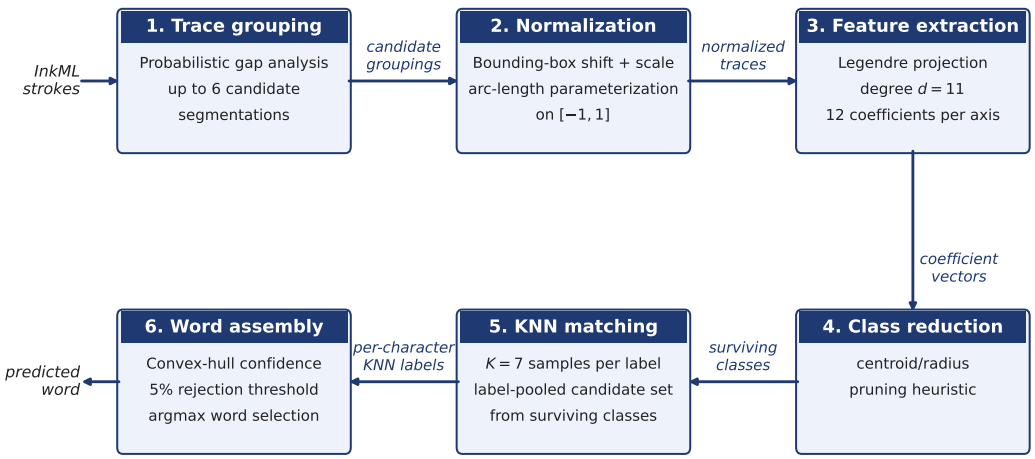

Figure 6.1: Recognition pipeline. InkML strokes enter on the left; the predicted word leaves on the right.

The rest of the chapter goes through each stage in order. For every stage it states the problem, explains the algorithm, writes out the relevant formulas, and lists the parameters used. A single-page pseudocode summary of the whole pipeline is in Appendix 10.

6.1 Probabilistic trace grouping

6.1.1 The segmentation problem

A handwritten word is made up of a sequence of pen strokes. However, the classifier processes each character individually. Character-by-character processing was covered in the studies by Tappert et al. (1990) [67] and Plamondon and Srihari (2000) [1]. To begin the process of character classification, the system must determine which strokes make up the same character. For example, the letter *i* consists of a vertical line followed by a dot. As another example, the letter *t* typically includes a vertical line with a horizontal bar underneath it. Therefore, a word such as *it* contains four individual strokes; however, there are only two characters.

This task of determining which group of strokes corresponds to the same character is commonly referred to as *segmentation* or *grouping*. The problem with segmentation is that both the correct grouping of the strokes into characters and the determination of which characters correspond to those groups depend upon each other. In other words, the correct grouping cannot occur until after the characters have been determined; however, neither the correct grouping nor the correct characters can be determined independently.

Casey and Lecolinet(1996) [3] review various techniques used in segmenting characters from handwritten texts, and Bunke(2003) [64] reviews segmentation-based approaches to recognizing Roman cursive script and segmentation-free approaches to recognition of Roman cursive script. This is the form of Sayre's paradox that the segmentation-then-recognition tradition tries to break.

The pipeline here avoids committing to one grouping. It generates several candidate segmentations instead, ranks them by a probabilistic score, and passes all of them through to the classifier. If the top-ranked segmentation is wrong, the correct one may still show up further down the list. In practice up to six candidates are generated per word.

6.1.2 Gap analysis

The algorithm starts by measuring how far apart every pair of strokes is. The gap loop runs over all unordered stroke pairs (i, j) with $i < j$, not only over temporally adjacent pairs. Impossible non-local joins are filtered out by the certain-separate threshold: any pair whose normalized gap exceeds 0.5 yields a join probability below 0.1 and is excluded from joining. For strokes i and j, the raw gap g_{ij} is the smallest weighted Euclidean distance between any point on stroke i and any point on stroke j:

$$g_{ij} = \min_{p \in t_i,\, q \in t_j} \sqrt{(p_x - q_x)^2 + (\alpha_v \cdot (p_y - q_y))^2},$$

where $\alpha_v = 0.2$ is a vertical weight. The weight is less than 1 because Latin script writes characters side by side along a horizontal baseline. A big horizontal gap between two strokes is a strong signal that they belong to different characters. A big vertical gap is a much weaker signal: the dot of i sits well above the vertical stroke and the two are still part of the same character. The factor $\alpha_v = 0.2$ shrinks the vertical contribution by a factor of five, so a vertical gap of 50 units carries the same weight as a horizontal gap of 10 units.

The raw gap is normalized by a *natural scale* $\bar{s}$, defined as the average maximum bounding-box dimension across all strokes in the word:

$$\hat{g}_{ij} = \frac{g_{ij}}{\bar{s}}, \qquad \bar{s} = \frac{1}{N} \sum_{k=1}^{N} \max(\text{width}_k, \text{height}_k).$$

This normalization puts gap values on a common scale across words of different sizes. A gap of $\hat{g} = 0.5$ means the strokes are half a character-width apart, whether the word was written large or small.

6.1.3 Join probability model

Each normalized gap is converted to a join probability using a sigmoid function:

$$p(\text{join} \mid \hat{g}) = \frac{1}{1 + \exp\big(\kappa \cdot (\hat{g} - \mu)\big)},$$

where $\kappa = 10$ controls how steep the curve is and $\mu = 0.3$ is its midpoint. The sigmoid sends small gaps (strokes close together) to probabilities near 1, and large gaps to probabilities near 0. The midpoint $\mu = 0.3$ means a normalized gap of 0.3 character-widths gives a 50/50 join probability. Gaps below 0.15 sit above 0.95; gaps above 0.5 sit below 0.05. The curve is plotted in Figure 6.2.

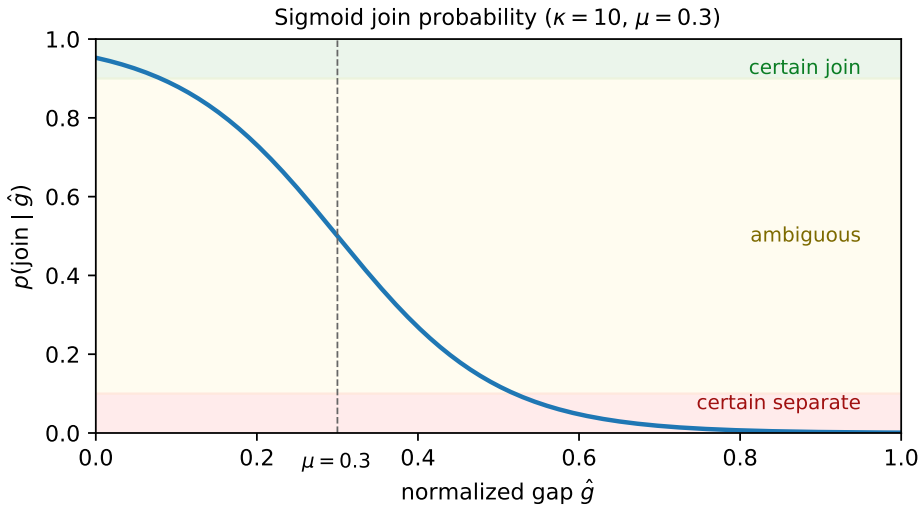

Figure 6.2: Sigmoid join probability $p(\text{join} \mid \hat{g})$ with $\kappa = 10$ and $\mu = 0.3$.

Three heuristics override or modify the sigmoid output for special cases:

Touching strokes. If any point on stroke i is within 2.0 coordinate units of any point on stroke j, the strokes are treated as touching. Touching strokes receive a join probability of 1.0 regardless of the sigmoid. This handles separately-recorded strokes whose endpoints or sampled points touch.

Dot strokes. An extremely small stroke whose bounding box is less than $0.3 \times \bar{s}$ in both width and height is classified as a *dot* (as in i or j), which nearly always belongs to the preceding or succeeding character. Therefore, the normalized gap value for a dot stroke is reduced by a factor of 0.3 prior to being passed through the sigmoid function. Hu and Zanibbi [46] used a similar method of identifying dots, minus signs, and lines from fractions at the beginning of each line segment. Multiplying the dot's normalized gap by

0.3 decreases its size and increases the likelihood that the dot will be classified as part of the preceding character.

Stacked strokes. Some characters contain a shorter stroke on top or below another longer stroke. For example, the dot of *i*, the crossbar of *T*, and the two bars of =. The stacked-stroke heuristic will pick out those characters. If there is a smaller stroke (for example, a dot, or a horizontal bar in which the width-to-height ratio is greater than 3) whose center falls within the horizontal span of a longer stroke, and if the distance vertically from the centers of both strokes is less than 1.2× the length of the longest dimension of the longer stroke, the join probability is raised to at least .95. If the vertical separation is less than 0.8× that dimension (the small stroke sits tight against the body), it goes up to 0.99.

6.1.4 Gap categorization

Once the join probability is in hand, each pairwise gap falls into one of three categories:

- **Certain join** ($p > 0.9$): the two strokes belong to the same character with high confidence. The system commits to joining them in every candidate segmentation.
- **Certain separate** ($p < 0.1$): the two strokes belong to different characters with high confidence. The system never joins them.
- **Ambiguous** ($0.1 \leq p \leq 0.9$): the evidence is mixed. Different candidates will make different choices about these gaps.

The thresholds 0.9 and 0.1 are the values used by the implementation; the consolidated parameter table in Chapter 7 lists them alongside the other pipeline constants.

The certain joins and certain separates form the backbone of every candidate. The ambiguous gaps are what make one candidate differ from the next.

6.1.5 Candidate generation

The categorized gaps are used to build up to six candidate segmentations, each one a different guess about how the ambiguous strokes should be grouped. This over-segmentation-and-rank strategy is the segmentation-based path covered by Casey and Lecolinet [3] and by Bunke [64]. Figure 6.3 shows two such groupings of the same input.

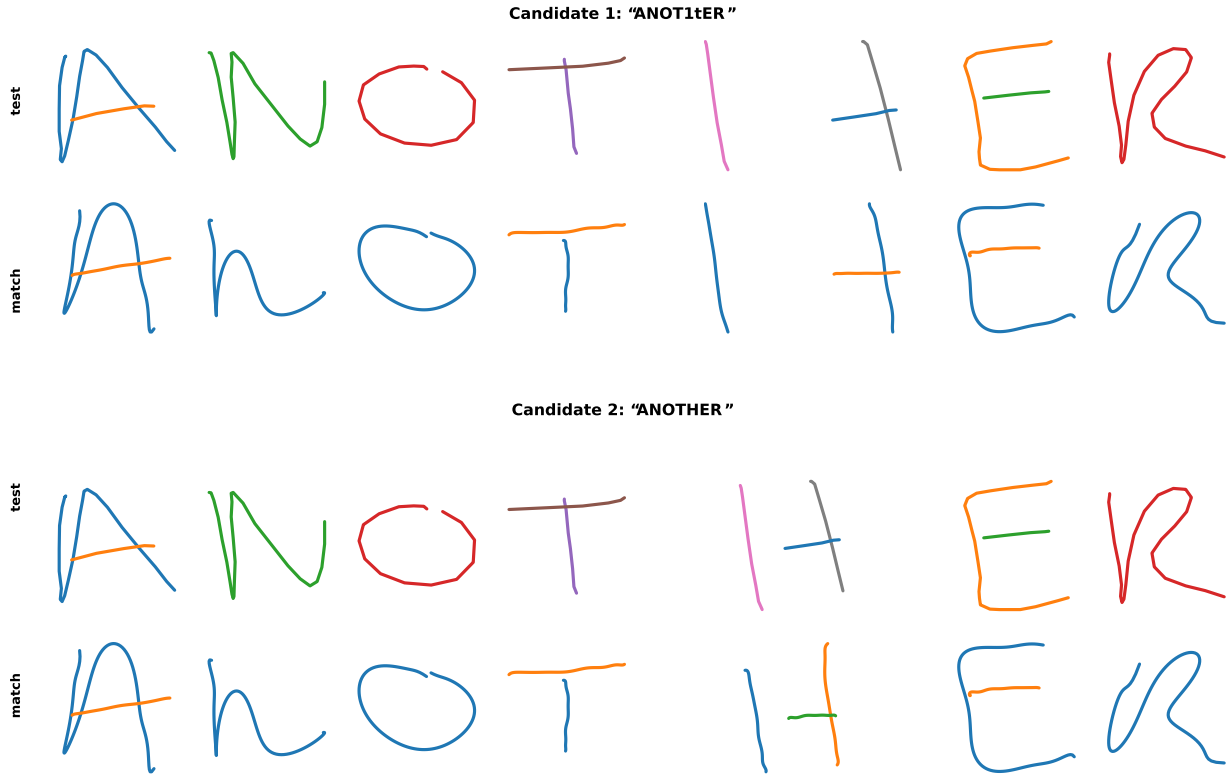

Figure 6.3: Two candidate groupings of the same input word, a UniPen sample of `ANOTHER`. Each column shows the test strokes assigned to one character (*test*) above the database sample matched by k-nearest neighbours (*match*). Stroke colour is fixed by stroke index, so the same physical stroke carries the same colour in both candidates. Candidate 1 splits the `H` into a separate `1` and `t` and produces the wrong reading `ANOT1tER`. Candidate 2 keeps the three strokes of the `H` together and produces the correct reading `ANOTHER`.

Grouping by merging. Every candidate starts with each stroke in its own single group. Joins are then applied by merging groups: if strokes 1 and 3 are joined, the group containing stroke 1 absorbs the group containing stroke 3. The same step is repeated for every join in the candidate's join set.

Baseline candidate. The first candidate applies every certain join and none of the ambiguous ones. This gives the most conservative segmentation, splitting the word into the largest number of characters consistent with the high-confidence evidence.

Progressive candidates. The ambiguous gaps are sorted by join probability, most probable first. The second candidate adds the single most probable ambiguous join on top of the baseline; the third adds the next one as well; and so on until six candidates have been generated or the ambiguous gaps run out.

Deduplication and scoring. Different join sets can lead to the same grouping. If strokes 1, 2, and 3 are all close together, joining (1,2) then (2,3) ends up with the same group as joining (1,3) then (1,2). Duplicate groupings are removed after generation. Each remaining candidate is given a score equal to the average join probability across all within-group stroke pairs:

$$\text{score} = \frac{1}{|\mathcal{P}|} \sum_{(i,j)\in\mathcal{P}} p(\text{join} \mid \hat{g}_{ij}),$$

where $\mathcal{P}$ is the set of all stroke pairs that share a group in the candidate. The score is well-defined only when $\mathcal{P}$ is non-empty; single-stroke groups contribute no within-group pair, so the score defaults to 1.0 when no within-group pair exists. This default favours all-singleton segmentations at the ranking stage, but the downstream classifier rejects most of them on confidence so the effect on the final word prediction is small. The score is used *before* classification, to rank the candidates that are then passed to the classifier, not as a post-classification reranker. A high score means the candidate is joining pairs that the probability model already wanted joined. Candidates are then sorted by this score in descending order.

6.1.6 Per-candidate normalization

Each character group in each candidate then goes through the same normalization and feature extraction described in Chapters 3–4. The strokes in the group are enclosed in a bounding box, shifted and scaled, reparameterized by arc length, and projected onto the Legendre basis. Direction canonicalization is applied. The result is a single flat coefficient vector of dimension $24n$, where n is the number of strokes in the group. This vector is what the classification stage takes as input.

6.2 k-nearest-neighbour classification

The trace grouping stage returns, for each candidate segmentation, a list of character groups with their coefficient vectors. The job of the classification stage is to take each of those vectors and find the closest character label in the reference database (Chapter 5). Golubitsky and Watt [7] worked out distance-based classification on functional coefficient vectors for isolated handwritten symbols and reported around 97.5% accuracy with a two-stage method built on the Euclidean distance to the convex hull of nearest neighbours.

The database has over 3,000 classes and 76,000 samples, so comparing the test vector against every sample is prohibitively expensive at interactive rates: a full sweep would require on the order of 7.6×10^4 weighted-distance evaluations per character, which is two orders of magnitude above what a single-threaded recognizer can sustain inside a sub-second per-word budget. Classification is split into two steps. The first is a fast *class reduction* step that throws away most of the database from class-level statistics alone. The second is a *sample-level KNN* step on the classes that survive. Two-phase prototype-based schemes of this shape have shown up before: Vuori et al. [23] reduce and reorder a prototype set in a first pass and run the full similarity measure in a second, and Bahlmann and Burkhardt [22] take a similar cluster-then-classify route inside their CSDTW system.

6.2.1 Distance metric

Every distance in the classification pipeline is a weighted Euclidean distance, the standard choice for vector-space pattern classification covered in Duda and Hart [73]. For two flat coefficient vectors $\mathbf{a}$ and $\mathbf{b}$ that describe characters with n strokes at polynomial degree $d = 11$:

$$d(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{s=1}^{n} \sum_{j=0}^{d} w_j \left[(a_{s,j}^{x} - b_{s,j}^{x})^2 + (a_{s,j}^{y} - b_{s,j}^{y})^2 \right]},$$

where s indexes strokes, j indexes the Legendre coefficient within each stroke, and w_j is the weight for degree j.

The basis is the ordinary orthogonal Legendre basis (Chapter 4); the weighting scheme determines the inner product used for the distance, not the basis itself. Two weighting schemes are used in this thesis:

L2 curve-distance weights. The weights $w_j = 2/(2j+1)$ make the coefficient-space distance equal to the L^2 distance between the reconstructed curves (Section 5.8). Under this scheme the zeroth coefficient (average position) has weight 2.0, the first coefficient (slope) has weight 0.67, and the eleventh (fine oscillation) has weight 0.087. Low-order shape dominates and high-order detail barely registers. These weights are used only for the class radius computation, where the radius is the 90th-percentile L^2 curve distance from a sample to its class centroid.

Sobolev weights. The weights $w_j = 1 + \alpha j^2$ with $\alpha = 0.3$ grow with the degree. The zeroth coefficient has weight 1.0, the fifth has weight 8.5, and the eleventh has weight 37.3. The scheme puts much more weight on the mid- and high-order coefficients that encode curvature, bends, and fine shape differences. Mazalov and Watt [12] show that tuning the Legendre–Sobolev jet scale sharpens the gap between characters whose strokes sit in similar places but curve in different ways. The letters n and m, for example, have similar bounding boxes and stroke positions but a different number of arches, and the Sobolev weights amplify exactly that kind of curvature difference. The Sobolev scheme is used for every distance computation during classification: centroid-based class reduction, sample-level KNN, and convex hull ranking. Chapter 8 reports the resulting per-character accuracy on the test set.

6.2.2 Stroke permutation search

A multi-stroke character can be drawn with the strokes in any order. One writer might draw the left leg of H first. Another might start with the right leg. The order recorded in the input does not have to line up with the order stored in a database sample. Golubitsky and Watt [11] run into the same stroke-order ambiguity in their multi-stroke symbol recognizer.

The distance function deals with this by trying every ordering. For a permutation π, stroke $\pi(s)$ of the test character is matched against stroke s of the database sample. The distance is computed for each permutation, and the smallest one is kept:

$$d_{\min}(\mathbf{a}, \mathbf{b}) = \min_{\pi \in S_n} d(\mathbf{a}_\pi, \mathbf{b}),$$

where S_n is the set of all permutations of n elements and $\mathbf{a}_\pi$ is the test vector with its stroke blocks reordered according to π.

For $n = 1$ (single-stroke characters, 69% of the database) there is only one ordering and no search is needed. For $n = 2$ there are 2 permutations, for $n = 3$ there are 6, and for $n = 4$ there are 24. Characters with more than four strokes are rare.

Adjacency-based filtering. Some orderings are implausible. Stroke orderings inconsistent with the writer's stroke-spatial proximity are unlikely; the search prunes orderings whose adjacent strokes are more than 0.4× the maximum coordinate range apart. The system creates permutations as before. Then the system determines whether each ordered sequence of consecutive strokes has its endpoints too far apart. In particular, for each pair of consecutive strokes, the system calculates the length of the vector from the end point of the first stroke to the start point of the second. If any of these vectors have lengths greater than 0.4 times the maximum coordinate range, the entire permutation is discarded.

However, if no permutations survive the above filter, all permutations are kept and ordered by the minimum vector length. This fallback keeps the system from being left with no valid ordering for unusual stroke arrangements.

6.2.3 Centroid-based class reduction

The first classification step compares the test vector against the centroid of every class whose stroke count matches the test character. Wilson and Martinez [60] survey instance-set reduction techniques for nearest-neighbour classifiers, and the centroid-with-radius step used here plays the same role: cut the candidate set down to a manageable size before any sample-level comparison happens. Only classes with the same stroke count as the test character are looked at, so a two-stroke test character is never compared against one-stroke classes.

For each eligible class k, two quantities are computed. The centroid distance d_k is the weighted distance from the test vector $\mathbf{x}$ to the class centroid $\mathbf{c}_k$. The reduction score ℓ_k approximates how close $\mathbf{x}$ could plausibly sit to any member of the class, using the class radius r_k as a cutoff:

$$d_k = d(\mathbf{x}, \mathbf{c}_k), \qquad \ell_k = \max(0,\ d_k - r_k).$$

The form of ℓ_k is borrowed from the triangle inequality: if a class member sat at most r_k from the centroid under the same metric as d_k, then $d_k - r_k$ would be a true lower

bound on the distance from $\mathbf{x}$ to that member. The class radius r_k is computed under the L^2 curve-distance weights while d_k is computed under the Sobolev weights, so the two quantities do not share a metric and ℓ_k is not a mathematical lower bound. It is used here as a pruning heuristic, a lower cutoff that ranks classes for inclusion in the next stage rather than a guarantee on member distances. Golubitsky and Watt [7] reduce irrelevant classes in a comparable first stage of their two-stage Manhattan-then-hull recognizer before applying the more expensive convex-hull distance.

A *dual-score union* strategy then picks the classes that survive. Two sorted lists are formed:

1. The 60 classes with the smallest reduction score ℓ_k.
2. The 40 classes with the smallest centroid distance d_k.

The two lists are merged into one set, sorted by centroid distance, and cut down to 50 classes.

The two-list approach is needed because neither ranking on its own is sufficient. Ranking by ℓ_k alone tends to favour classes with large radii (loose classes), which can yield small cutoff scores even when their centroids are far away. Ranking by centroid distance alone favours classes whose centroids are close to $\mathbf{x}$, but it can miss tight classes whose centroids sit slightly farther out even though their members are close. Taking the union catches both cases.

6.2.4 Sample-level KNN

Class reduction leaves about 50 candidate classes. The KNN step then computes the distance from the test vector to every individual sample inside those classes.

The distances are bundled by character label. All samples labelled A (no matter which class they came from) go into one group, all B samples into another, and so on. Inside each label group the samples are sorted by distance and the $k = 7$ nearest are picked. The nearest-neighbour rule itself goes back to Fix and Hodges [79] and was analysed by Cover and Hart [80]; Aha, Kibler, and Albert [59] reframe it as an instance-based learning algorithm and study how it behaves on noisy training data.

Grouping by label rather than by class is a deliberate choice. The natural alternative is to group by class: take the k nearest samples from each *class* and keep the class structure

intact. Grouping by label instead lets a rare class with only 3 samples still help a label's hull estimate alongside a common class with 50 samples. Label grouping also stops any one label from occupying a disproportionate share of the candidate pool just because it happens to span many classes. A label with 100 classes and 1,000 samples ends up with the same $k = 7$ representatives as a label with 5 classes and 30 samples.

The k nearest samples for each label are used for two things:

1. Their average distance gives a KNN score for the label.
2. Their coefficient vectors are passed to the convex hull computation (Section 6.3) for a geometric ranking.

6.2.5 Digit-label normalization

The pairs *0/O*, *1/l*, *5/S*, and *9/g* are visually indistinguishable in isolation. To prevent the k-nearest-neighbour algorithm from artificially splitting nearest neighbours across these visually-equivalent labels, each pair is merged into a single pooled label *for KNN candidate retrieval only*. The mapping is applied before the label-pooling step inside KNN: every sample with one of these labels is treated as belonging to the merged label group when the seven nearest neighbours per label are selected. The original labels are preserved in the database; nothing is rewritten on disk. Evaluation then uses the merged label for the superclass-corrected metric and the original label for strict character matching, so the merge changes which neighbours feed the convex-hull computation but does not change how the database is stored or how strict accuracy is scored. Note: the 9/g pair is merged here for KNN retrieval; the broader 9/q/g superclass group used for scoring is defined separately in Section 7.3.

6.3 Convex hull ranking and confidence

For each candidate label the KNN stage gives back the $k = 7$ nearest database samples together with a KNN score, which is just their average distance. That score is a fine first cut, but it treats each of the k neighbours as a separate point. It says nothing about whether the test character actually sits inside the region of coefficient space that those neighbours cover.

Picture two labels whose 7 nearest samples are at the same average distance from the test character. For one label the 7 samples sit around the test character on all sides. For the other they cluster on one side and the test character sits off to the other side. The KNN score is identical, but the first label is the better match: the test character is inside the region that the label's samples define. Vincent and Bengio [61] make this geometric argument in their k-local Convex Distance Nearest Neighbour algorithm and show that the convex hull of the per-class k-neighbourhood is a better local model than the k neighbours treated as independent points.

Convex hull ranking captures that distinction. For each label, the k samples are taken as the vertices of a simplex in coefficient space, and the distance from the test character to that simplex is computed. Vincent and Bengio [61] solve the same problem with quadratic programming on USPS and MNIST digits, and Golubitsky and Watt [7] apply the construction to functional-coefficient vectors on 50,703 handwritten mathematical symbols across 242 classes. They report that distance to the convex hull of nearest neighbours gives the best classification accuracy of the methods they compared. Here the hull distance replaces the KNN score as the ranking criterion. Figure 6.4 shows two competing classes ranked this way.

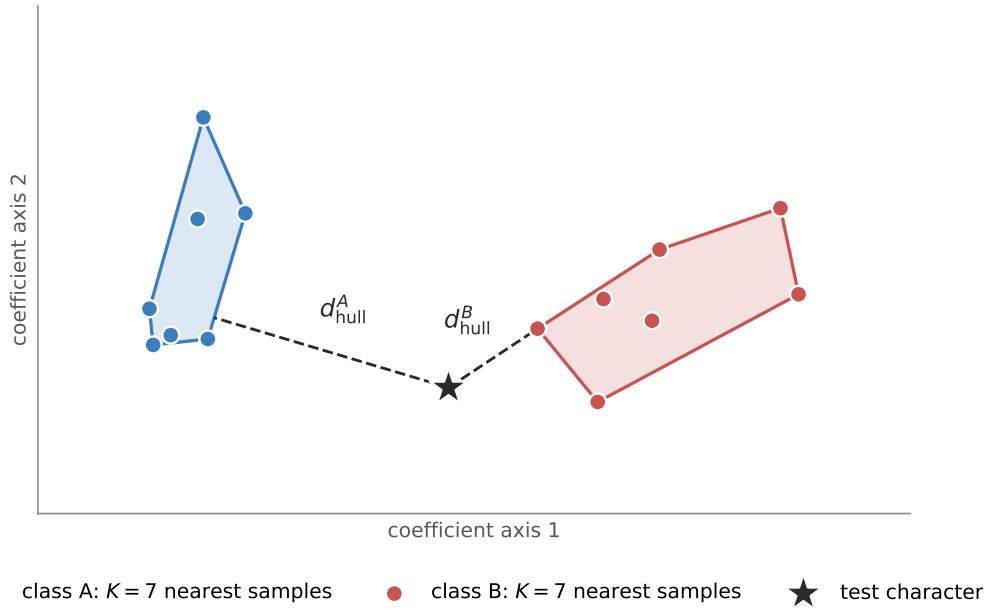

Figure 6.4: Convex-hull distance schematic. Two competing classes A and B, each with its seven nearest samples and their convex hull; the dashed perpendiculars give the hull distances d^A_{hull} and d^B_{hull} from the test character.

6.3.1 Simplex distance computation

Given the k nearest samples $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ for a label, the hull distance is the smallest distance from the test vector $\mathbf{x}$ to any point that can be written as a convex combination of the k vertices:

$$d_{\text{hull}}(\mathbf{x}) = \min_{\substack{\lambda_i \geq 0 \\ \sum_i \lambda_i = 1}} \left\| \mathbf{x} - \sum_{i=1}^{k} \lambda_i \mathbf{v}_i \right\|_2 .$$

If $\mathbf{x}$ lies inside the convex hull, the distance is zero (or close to zero up to numerical precision). If it sits outside, the distance is how far outside. The same quadratic-programming formulation is given by Vincent and Bengio [61].

The computation is done by a recursive algorithm with three levels:

$k = 1$ **(point).** The hull is a single point. The distance is the standard Euclidean distance $\|\mathbf{x} - \mathbf{v}_1\|_2$.

$k = 2$ **(line segment).** The hull is the line segment from $\mathbf{v}_1$ to $\mathbf{v}_2$. The test point $\mathbf{x}$ is projected onto the line through the two vertices. If the projection parameter t satisfies $0 \leq t \leq 1$, the closest point on the segment is the projection and the distance is the perpendicular distance. If $t < 0$, the closest point is $\mathbf{v}_1$; if $t > 1$, the closest point is $\mathbf{v}_2$.

$k \geq 3$ **(general simplex).** The barycentric coordinates $\lambda_1, \ldots, \lambda_k$ come from solving the linear system that writes $\mathbf{x}$ as an affine combination of the vertices. The system is solved by Gaussian elimination with partial pivoting. If every $\lambda_i \geq 0$, the point $\mathbf{x}$ already sits inside the simplex and the distance is the norm of the residual after projection. Mazalov [6] gives the same projection-onto-simplex recipe in his thesis and reports its use for online symbol classification.

If some $\lambda_i < 0$, the closest point on the simplex sits on one of its faces. The algorithm picks the vertex $\mathbf{v}_m$ with the most negative λ_m, builds every $(k-1)$-dimensional face that still includes the vertex closest to $\mathbf{x}$ (the one with the largest λ), and recurses on each face. The smallest distance across all recursive calls is returned. Golubitsky and Watt [7] describe the same simplicial decomposition for handling faces in their convex-hull classifier.

For $k = 7$ in a 24-dimensional space (single-stroke characters) the linear system at each recursion level is small and the computation is fast.

6.3.2 Coordinate alignment

Each of the k nearest samples had their corresponding test character compared by using a specific stroke permutation as described within Section 6.2.2 for KNN. The distance was calculated based on this permutation. The sample's coefficient vector was already in the database in the original stroke order.

Before constructing the hull, each sample's coefficient vector must be reordered into the permutation that resulted in its best distance. For example, if the best match for a sample used test stroke 1 with sample stroke 3 and test stroke 2 with sample stroke 1,

then the sample's 24 coefficients would have to be interchanged such that sample stroke 3 comes before sample stroke 1.

This reordering is necessary. Otherwise, the hull would be constructed from vectors describing strokes in different orders. Therefore, the addition of two vectors that represent different strokes does not correspond to a valid point.

6.3.3 Confidence and rejection

The hull distance for each label is turned into a confidence score using a label-specific radius r derived from the class radii (Section 5.8). Because KNN is label-pooled, the seven nearest neighbours for a given label may come from several different underlying classes; the radius used is $r = \max_k r_k$ over all classes k that contributed at least one sample to the k-nearest neighbours for that label. Golubitsky and Watt [81] developed confidence measures for handwritten mathematical symbol recognition in the same coefficient-based framework:

$$\text{conf} = \max\left(0,\ \min\left(1,\ 1 - \frac{d_{\text{hull}}}{r}\right)\right).$$

The formula maps hull distance to a number between 0 and 1:

- $d_{\text{hull}} = 0$ (test character inside the hull): confidence is 1.0.
- $d_{\text{hull}} = r$ (test character is one class-radius away): confidence is 0.0.
- $d_{\text{hull}} > r$: confidence is clamped to 0.0.

The radius r is computed under the L^2 curve-distance weights while d_{hull} is computed under the Sobolev weights, so the confidence formula is used as a heuristic implementation choice rather than a metrically rigorous normalization.

The confidence score is computed for each surviving label, and the label with the maximum confidence is chosen as the predicted character. Labels are sorted by confidence in descending order and the top 8 are kept for each character position. If even the top confidence drops below 5%, the character is rejected and labelled *?* in the output. A rejected position is one where nothing in the database is a plausible match.

6.3.4 Size-based distance adjustment

A special case shows up with tiny punctuation marks such as periods and commas. These characters have a very small bounding box in the raw input. Their coefficient vectors cluster in a tight region of the space and the class radius comes out small. A normal-sized letter tested against a punctuation class can end up with a low hull distance simply because the class occupies a small volume, not because the test character really looks like punctuation. Mazalov and Watt [13] hit the same problem with periods, commas, and other naturally-small mathematical symbols, and propose a size-based correction on top of the shape-only distance.

To fix this, the hull distance for a punctuation class is scaled up by a penalty that reflects the size mismatch between the test character and the class:

$$d_{\text{adjusted}} = d_{\text{hull}} \cdot \big(1 + \beta \cdot \min(r_{\log}, c_{\max})\big),$$

where $r_{\log} = |\log(\text{test size}/\text{class avg size})|$ is the log-ratio of the test character's bounding-box size to the class's average pre-normalization size, $\beta = 1.2$ is the penalty strength, and $c_{\max} = 2.5$ caps the penalty so it cannot blow up on extreme ratios. The adjustment runs only on classes labelled "." or "," whose average pre-normalization size is below 30% of the global average across all classes. All other classes are unchanged.

6.4 Word assembly

By the end of the three classification stages, every character position in every candidate segmentation has a predicted label and a confidence score. The per-character labels are concatenated into a predicted word for each candidate:

$$\hat{w}^{(c)} = \hat{y}_1^{(c)}\ \hat{y}_2^{(c)}\ \cdots\ \hat{y}_{M_c}^{(c)},$$

where M_c is the number of character groups in candidate c and $\hat{y}_i^{(c)}$ is the predicted label for the i-th group.

Each candidate gets a word-level confidence equal to the mean of its character confidences, and the final prediction is the candidate word chosen by mean character confidence

(an argmax over candidates); this is not dynamic programming. The system outputs that word together with the top 8 character-level alternatives at each position. Zimmermann and Bunke [82] apply n-gram language models to rescore offline handwritten text recognition output of this shape, and Carbune et al. [29] use a character language model in the same role inside Google's production online recognizer.

Chapter 7

Experimental design and evaluation protocol

7.1 Overview

This chapter sets out how the recognition pipeline in this thesis was evaluated. It covers four things: the test words, the rules for scoring predictions, the hardware and software the pipeline runs on, and the fixed values of every numerical parameter used during the evaluation.

The test words come from the UniPen word collection, filtered down to the 62-character alphabet that the reference database of Chapter 5 works with. The test set is also kept clear of the database itself: no word used for evaluation contributed any sample to training, and the separation holds at the source-file level.

Evaluation accuracy is provided at two levels. For individual words, a prediction will be counted toward overall performance only if the predicted entire string is identical to the known correct response. For individual characters, accuracy will be determined based upon their respective positions within the strings (with some flexibility afforded for both capitalization differences and for digit-letter pairs where characters appear the same). In addition to reporting the number of times the correct answer was found in the top-1 (highest-ranking) list of possible responses, top-2 and top-3 retrieval rates are reported alongside the top-1 numbers, so it is clear how often the correct answer sits among the pipeline's near-misses rather than being absent altogether.

7.2 Test set

The test set is drawn from the UniPen word collection, a public database of online handwriting collected from many writers on several input devices, introduced by Guyon et al. (1994) [62] and reviewed in the online handwriting survey of Plamondon and Srihari (2000) [1]. UniPen covers isolated characters, words, and sentences; only the word portion is used here.

The raw collection contains words with punctuation, accented letters, and non-Latin symbols. None of these fall inside the alphabet the reference database covers. Running the pipeline on a word that contains, for example, an apostrophe or a ç would mix up a recognition question with a coverage question, because there is simply nothing in the database for such a character to match against. To keep the two questions apart, the test set is filtered down to the 62-label Latin-plus-digits alphabet (`A--Z`, `a--z`, `0--9`), and any word that contains a character outside this set is dropped. The filter removes diacritics, punctuation, and symbols while preserving the cursive and printed styles that UniPen captures for the alphabet itself. Vuurpijl et al. (2004) report that the UniPen devset contains both labelling and segmentation errors [63], which is part of why the filtering here is on the conservative side.

After filtering, the test set contains 3,154 handwritten words. Each word is stored as an InkML file: a list of strokes in (x, y, t) form, together with the correct answer for that word recorded as a label annotation inside the file. Word lengths range from two characters up to the 26 characters of a full alphabet sample. Table 7.1 summarizes the content.

Property	**Value**
Total words	3,154
Mean word length	5.5 characters
Median word length	5 characters
Shortest word	2 characters
Longest word	26 characters (full alphabet samples)
Unique writers	Multiple (UniPen-provided)
Character label set	62 (A–Z, a–z, 0–9)

Table 7.1: Summary of the filtered UniPen word test set.

The reference database in Chapter 5 was built from isolated-character UniPen samples and from stroke-level CROHME extractions [40]. These are different source collections from the UniPen word material, and no word in the test set contributed any sample to the database. The separation is enforced at the source-file level, so no fragment of a test word slips into training even when the same character happens to appear elsewhere in isolation. The label-level coverage of the database is uneven (Figure 7.1): common lowercase letters like *e*, *a*, and *o* have thousands of samples each, while rare uppercase letters like *Q* and *Z* have only a few hundred. This imbalance shows up again in the per-character accuracy reported in Chapter 8.

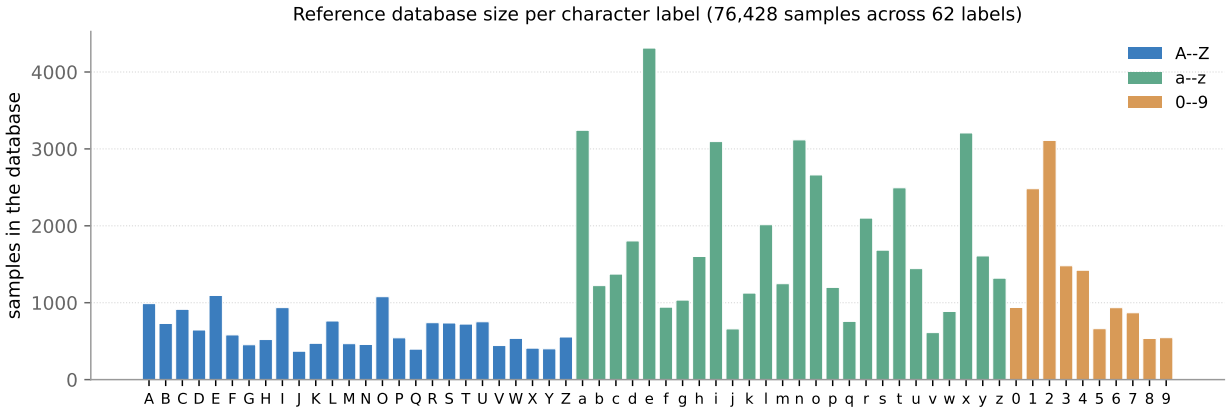

Figure 7.1: Reference database size per character label, by category: A–Z (blue), a–z (green), 0–9 (orange).

7.3 Evaluation metrics

Two top-1 accuracy rates are reported:

- **Word-level top-1 accuracy:** the fraction of test words whose top-1 predicted string matches the correct word exactly.
- **Character-level top-1 accuracy:** the fraction of character positions whose top-1 predicted label matches the correct label, computed only over words whose predicted length equals the correct length.

It is common to provide both rates when evaluating a handwriting recognition system. Many studies have reported both character and word error rates, including Bunke (2003) [64], Graves et al. (2009) [25] and Carbune et al. (2020) [29]. Both types of measures capture different aspects of performance. For example, if there was an incorrect character in a single six-letter word, this would be counted as a word error but there would be five correct positions and one incorrect position for the character error rate.

The classification of words into their respective classes is done in a case-insensitive manner. In other words, since a sample labeled C and a sample labeled c could produce the same written shape, they will be considered part of the same class regardless of whether or not they are referred to as such. Since the pipeline has no information regarding capitalization, it cannot differentiate between them based on the input provided. Also, since the evaluation metric does not require differentiation between upper-case and lower-case letters, it does not penalize the pipeline for failing to do so.

The superclass rule grants an additional permission for matched labels based upon how they appear in handwriting. Several characters within the 62-label set can be confused based upon appearance. In order to determine which character is intended by the user, Yaeger et al. (1996) have shown that the Apple Newton Recognizer accepts ambiguity when searching for answers in context [83]. An example of ambiguity exists for a handwritten O and a handwritten *0*; each represents a closed oval. Another example is a handwritten l or a handwritten *1*, each representing a single vertical line. If either of these answers were marked incorrect, then the system would be penalized for making a distinction that was not made by the input data. These ambiguities include three types: digit-letter group (digits written like letters), letter-letter group (shapes for letters that result in nearly identical handwritten traces), and vertical-line confusion group (capital I, digit *1*, lowercase i, and lowercase l).

Vinciarelli (2002) has stated that the problem described above relates to Sayre's Paradox; i.e., letters cannot be properly segmented until they are identified, and many cursive script symbols cannot be properly identified independently [66].

A character position counts as correct whenever the exact label matches, or whenever the predicted and correct labels fall in the same group. The rule applies to character-level scoring only. Word-level exact matching ignores it. The word-accuracy-with-superclass row in Table 8.1 applies the superclass rule character-by-character (each predicted position is correct if its label or any superclass-equivalent label matches), then declares the word

Kind	Group	Why they look alike
Digit-letter	0, O, o	Closed oval
Digit-letter	5, S	Open S-curve
Digit-letter	9, q, g	Circle with a descender
Letter-letter	U, V	Curved vs pointed bottom
Vertical stroke	I, 1, i, l	Single vertical stroke

Table 7.2: Superclass groups used for character-level scoring. Only pairs whose handwritten shapes are routinely indistinguishable without surrounding context are retained.

correct if all positions are correct. This is a separate score from the strict exact-match word accuracy.

The superclass groups in Table 7.2 go further than this classification-time merge. They also cover pairs where the classifier is still asked to pick a single answer, but where a human reader of the written word would not be able to say which letter was intended without looking at the surrounding context.

Character-level scoring is only defined when the predicted string has the same length as the correct word. When the lengths differ, the failure is a segmentation failure: the trace-grouping stage produced the wrong number of characters, and there is no natural way to line predicted positions up with correct ones. Chapter 8 counts those cases separately and keeps them out of the character-level total. Segmentation failures and character confusions come from different parts of the pipeline, and they need different fixes.

The pipeline also keeps its top-2 and top-3 candidate strings, ranked by combined confidence. Carbune et al. (2020) report top-1 character and word rates from a CTC beam-search decoder that scores N-best candidates by combined log-probability [29], and the cursive survey of Bunke (2003) describes reordering an N-best list as a standard post-processing step [64]. The top-k retrieval rate is the fraction of test words for which the correct word appears anywhere in the top k predictions. A gap between top-1 and top-3 accuracy is a ranking problem, not a recognition problem: the classifier did produce the right answer, just not in first place.

7.4 Implementation

All experiments ran on an Apple M4 Pro chip with 24 GB unified memory, in a 16-inch MacBook Pro (November 2024), running macOS Sequoia 15.3.1. Runs are single-threaded. No GPU is used. The reference database takes under 100 MB in memory, and the heaviest per-word operation is a bounded set of coefficient comparisons, so a single CPU core is enough to keep up.

The recognition program is written in C++ and compiled with GNU g++ version 15. The whole program compiles as a single translation unit. Underneath the program is a C++ framework that the author's supervisor provided at the start of the project. The framework supplies the primitives the rest of the pipeline depends on: an InkML parser, the orthogonal Legendre polynomial basis and its integration routines, and the low-level normalization utilities described in Chapter 3. The work in this thesis builds on top of that framework. The results were evaluated using a Python script which runs the recognition program on every test word and writes the output into an HTML report.

7.5 Parameter configuration

The recognition pipeline has a small number of numerical parameters spread across its stages. Table 7.3 lists each parameter together with the value it held during the evaluations in Chapter 8.

The trace-grouping parameters control how strokes are assembled into characters, as described in Chapter 6, Section 6.1. The vertical gap weight, the sigmoid steepness and midpoint, and the two certainty thresholds were all picked empirically. The segmentation-candidate budget of six is a compromise between two pressures: six candidates is enough for the classifier to recover from a bad grouping on the hardest words, and few enough that the per-word runtime stays under a few hundred milliseconds.

The normalization degree is fixed at eleven, which gives twelve Legendre coefficients per dimension per stroke. This follows the truncated orthogonal-series representation developed by Golubitsky and Watt (2008, 2010) and Mazalov and Watt (2012) [10, 7, 12]. Chapter 4 discusses why eleven is the right choice.

The distance parameters control how the classifier compares a test stroke against the database. The Sobolev-weighted scheme follows Mazalov and Watt (2012), who showed that a Legendre-Sobolev inner product outperforms a plain Legendre inner product for isolated-character classification [12]; the value $\alpha = 0.3$ used here is the value used by Mazalov and Watt (2012) for handwritten symbols [12] and is adopted here without re-tuning. The class reduction stage retains the top 50 classes by centroid distance, and the k-nearest-neighbour stage then draws up to seven neighbours per label from the full database. This two-stage reduction follows the distance-based pipeline of Golubitsky and Watt (2010) [7]. The rejection confidence threshold is set at 5%, applied post-hoc to the recognizer's output so that low-confidence per-character predictions are reported as "?" rather than forced onto the nearest centroid. The confidence signal itself is built from the convex-hull score, in the sense introduced by Vincent and Bengio (2001) [61].

Stage	**Parameter**	**Value**
Trace grouping	Vertical gap weight α_v	0.2
Trace grouping	Sigmoid steepness κ	10
Trace grouping	Sigmoid midpoint μ	0.3
Trace grouping	Certain-join threshold	0.9
Trace grouping	Certain-separate threshold	0.1
Trace grouping	Segmentation candidates per word	6
Normalization	Legendre degree	11 (12 coefficients)
Distance	Sobolev weight α	0.3
Class reduction	Top-N classes retained	50
Class reduction	Radius percentile	90
KNN	Neighbours per label k	7
KNN	Rejection confidence threshold	5%

Table 7.3: Parameter values used in the primary evaluation.

7.5.1 Parameter-selection protocol

No parameter in Table 7.3 was selected by maximizing accuracy on the UniPen word test set. Each value was fixed before the test set was scored, and the provenance of each value is as follows. The Legendre degree of eleven follows the truncated orthogonal-series representation of Golubitsky and Watt (2008, 2010) and Mazalov and Watt (2012) [10, 7, 12]. The

Sobolev weight $\alpha = 0.3$ is taken directly from Mazalov and Watt (2012) [12] and adopted without re-tuning. The top-N class reduction count of fifty and the neighbours-per-label $k = 7$ follow the two-stage distance-based pipeline of Golubitsky and Watt (2010) [7]. The trace-grouping parameters (κ, μ, the vertical gap weight α_v, and the certain-join and certain-separate thresholds) come from algorithm-design reasoning about the sigmoid mapping in Chapter 6, and were fixed on training-side intuition without reference to test accuracy. The segmentation-candidate budget of six is a design choice constrained by per-word runtime rather than by accuracy on the test set. The rejection confidence threshold of 5% is justified post-hoc on the recognizer's confidence distribution (Chapter 8), not by maximizing test-set accuracy.

7.6 Evaluation protocol

A single evaluation run is one pass over the test set. The recognition program is compiled first, from the source described in Section 7.4. The pipeline then walks through the 3,154 test words and runs the program once per word, handing it the test word, the reference database, and the segmentation-candidate budget. For each word, the program returns the ranked predicted strings, their combined confidence scores, and the per-character class assignments.

The pipeline compares each prediction against the correct word stored in the InkML file and applies the case-insensitive and superclass-aware rules of Section 7.3. The pipeline finishes by writing the per-word outcomes into a report that lists every word, its prediction, its correctness under each metric, and its ranked candidate list. Chapter 8 reads its numbers from that report.

The pipeline is deterministic at inference time. The same InkML file, run against the same database with the same parameters, produces the same output byte-for-byte, up to the floating-point determinism of the toolchain. The classifier's ranking and rejection logic is a function of the input and the database alone, and no seeded random draws happen at any stage. Any difference in per-word outcomes between two configurations therefore comes from the configuration change itself.

Chapter 8

Results, error analysis, and discussion

8.1 Overall performance

The question this chapter answers is straightforward. Given the stroke trace of a written word, how often does the pipeline give that word back? Chapters 3 through 6 built the pieces: grouping characters, projecting each grouping onto a Legendre basis and using Sobolev-weighted distances, KNN scoring against a database of pre-classified samples with a convex-hull distance, and selecting the candidate segmentation with the highest mean character confidence as the predicted word string. Chapter 7 fixed the configuration parameters and chose the data set. With those choices held, the pipeline runs once on each of the 3,154 words in the UniPen test set [62], and the output is compared against the labeled string.

The simplest is the fraction of words for which the output string matches the labeled word exactly, character for character: the *exact word match*. The second reports the fraction of individual characters predicted correctly, averaged across all words. The third widens the comparison from the single best predicted word to the top three. The pipeline produces up to six candidate segmentations and ranks them, and this third number is the fraction of words where the labeled answer appears among the first three candidates rather than exactly first. The fourth and fifth numbers repeat the word and character-level comparisons under a relaxed match rule. Visually-indistinguishable letter pairs, such as a written `1` against a written `l`, or a written `0` against a written `O`, are treated as the same symbol when comparing predicted to labeled. The exact rule is the superclass equivalence defined in Section 7.3. Table 8.1 collects the five figures for the primary configuration.

The character-level numbers sit higher than the word-level numbers because one wrong character anywhere in a word fails the whole word, while at the character level the same mistake costs one position out of the word's length. Graves et al. (2009) report both word error rate and character error rate as separate metrics for the same reason [25]. The

Metric	Value
Word accuracy (exact match)	39.1%
Word accuracy (with superclass)	44.2%
Word accuracy (top three predictions)	43.1%
Character accuracy (per word)	75.1%
Character accuracy (per word, with superclass)	83.6%
Character accuracy (per character)	83.8%
Character accuracy (per character, with superclass)	85.7%

Table 8.1: Recognition accuracy on the 3,154-word filtered UniPen test set.

size of the gap between the two depends on word length and on how the per-character errors distribute across words. A pipeline that scatters its errors across many words has a wider word-versus-character gap than one that concentrates them in a few badly-recognized words.

There are two rows for the accuracy of character prediction because the same per-character predictions can be averaged either across words (75.1%, every word weighted equally regardless of length) or across character positions (83.8%, longer words contributing more positions). The two definitions are set out in Section 7.3.

Each of the two character accuracy figures has a superclass-aware companion in Table 8.1. Counting visually-indistinguishable pairs as correct lifts the per-word figure from 75.1% to 83.6% and the per-character figure from 83.8% to 85.7%. The per-word average picks up the larger lift because each superclass rescue is a bigger fraction of a short word than of a long one, and short and long words carry equal weight in that average.

The same per-character classifier produces a usable answer beyond rank-1. The labeled letter sits in the top three of the KNN ranking 91.5% of the time and in the top five 93.5% of the time. Most of the per-character errors are within reach of a downstream re-ranker that has access to a few extra candidates per position [82, 24]. Section 8.2 comes back to this point when it splits the wrong-word total into four categories.

The gap between exact word match and the superclass-corrected variant comes from a small set of letter pairs whose pen traces look nearly identical even when the labels differ. The 5-percentage-point lift from 39.1% to 44.2% is dominated by the vertical-stroke group $\{\mathrm{I}, 1, \mathrm{i}, \mathrm{l}\}$. Section 8.2 reports how much of the strict-versus-superclass lift each group

contributes.

8.2 Error analysis

8.2.1 Where the wrong words come from

A wrong word can fail in more than one way, and each kind of failure points to a different fix. Every wrong word in the test set is placed in exactly one of four categories. The categories correspond to four different things that can go wrong, decided by comparing the predicted string against the labeled word. Table 8.2 reports the counts.

Category	**Count**	**% of wrong**
Trace-grouping error	735	38.2%
Recognition: superclass-equivalent	154	8.0%
Recognition: classifier fault	915	47.6%
Low-quality or difficult input	118	6.1%
Total wrong	1,922	100.0%

Table 8.2: Error categories on the filtered UniPen test set.

Most basic failures are caught by the first category. The predicted string has a different number of characters than the labeled word [3]. This occurs if trace grouping breaks the strokes down into too many or too few character units (character units are created prior to the classifier running). Although the classifier may perform reasonably well on each supplied group, it is being asked to classify the wrong groups. Of the 1,922 wrong words, 735 occur as such, 38.2% of total failures. As soon as the lengths differ, there is no obvious method of aligning the predicted position of each character with the corresponding position of its label. Therefore, word-level character accuracy scores are undefined, and thus only the word-level score will be recorded.

The remaining three classes all contain strings of correct length where every character can be paired with an equivalent labeled character. All that varies among these three classes is how the errors made at the individual character level are spread out.

The second class, *superclass-equivalent recognition*, collects words that are subject to mismatches on visually indistinguishable letters. These include pairs such as `I` vs. `l`, `O` vs. `0`, and `U` vs. `V`, which is described in Section 7.3. The classification rule uses superclass information to allow for equivalence across these visual groupings. This class contains 154 words, which contribute to the 5.1 percentage point increase from strict matching accuracy to superclass-aware accuracy shown in the overall performance metrics in Table 8.1.

The third category, *classifier fault*, collects words where at least one mismatch is not within a superclass group, but at least half of the characters in the word were recognized correctly. The input was readable to a human, the grouping was right, and the classifier still landed on the wrong letter for some position. Mazalov (2013) reports the same kind of single-character misfire in the isolated-symbol setting using the same family of features [6]. 915 words land here, the largest of the three same-length categories at 47.6% of the wrong total.

The fourth category, *low-quality or difficult input*, collects words where more than half of the characters were misclassified. When the majority of a word's characters come back wrong, the input itself is a far more plausible source of the failure than the classifier. The same classifier does not normally fail in correlated ways on five independent characters in a row. 118 words land here, 6.1% of the wrong total. The 50%-correct cut-off is a structural rule rather than a tuned threshold, applied uniformly across all word lengths. This is a heuristic diagnostic category rather than an independently verified ground-truth category, as no human review of the underlying input quality has been performed.

Two observations follow straight from the counts. First, low-quality or difficult input is the smallest of the four categories, not the largest. Most wrong predictions are not failures of input quality. Second, the two algorithmic categories, trace-grouping and classifier fault, together account for 86% of all wrong words: 38.2% from trace grouping and 47.6% from the classifier. The pipeline's exact-match ceiling is set by algorithmic choices in those two stages, not by the readability of the test inputs.

8.2.2 Worked examples

The four categories are easier to read against actual inputs. Each figure in this section is laid out in per-character columns. The top half of every column shows the writer's strokes for that character, with one color per pen stroke. The bottom half shows the database

sample the classifier picked for that character, drawn in the same color scheme. Below each column the hull distance and per-character confidence are reported, and the matched class is named above the bottom box. When the matched sample looks like the writer's strokes, the classifier did the most reasonable thing it could. When the two boxes differ, the failure sits on the algorithm. The expected and predicted strings appear in each figure's caption. Six examples are shown in total, one or two per category.

Grouping: over-segmentation. Figure 8.1 shows a case where the trace-grouping stage cut a single character into two. Hu and Zanibbi (2013) describe the same failure mode for multi-stroke symbols whose components are spaced apart in time [46]. The writer drew a printed capital `H` as three separate strokes: a left vertical, a right vertical, and a horizontal crossbar between them. The crossbar reaches the left vertical but does not quite touch the right vertical, so the gap between the right vertical and the rest of the H exceeded the within-character threshold. The grouping stage produced two character units instead of one, and the matched samples confirm what those two units became: the left vertical plus the crossbar matched a lowercase `t`, and the right vertical matched a lowercase `l`. The remaining letters `igh` were grouped and matched correctly. Five units reached the classifier where four were expected.

Figure 8.1: `High` → `tligh`. Trace-grouping over-segmentation: the capital `H` is split into a `t` and an `l` by a within-character gap.

Grouping: under-segmentation. Figure 8.2 shows the opposite failure, the mirror case to the under-segmentation of touching symbols catalogued by Hu and Zanibbi (2013) [46]. The writer drew the printed `t` with a long horizontal crossbar that runs over into the next character. The crossbar reaches into the body of the `h`, and the grouping stage treats the `t`-stem, the crossbar, and the `h`-stem as one connected character unit. The matched sample for the merged unit comes from the lowercase-h class, and only three character groups reached the classifier where four were expected. The lost `t` was lost upstream of any classifier decision.

Figure 8.2: `that` → `hat`. Trace-grouping under-segmentation: the long crossbar of the printed `t` bridges into the `h`, and the two characters are merged into one character group.

Recognition saved by the superclass rule. Figure 8.3 shows the `I`-versus-`l` ambiguity behind most of the superclass uplift. The capital `I` in the middle of `BRIBE` is drawn as a single vertical line. The closest database sample for that position is also a single vertical line, but it comes from the lowercase-l class. The two strokes are visually indistinguishable; the labels are not. The case difference between `R` and `r` is forgiven by case-insensitive matching, and the `I`/`l` substitution is forgiven by the superclass rule. Words like this come out wrong under exact-match scoring and right under the superclass-corrected scoring.

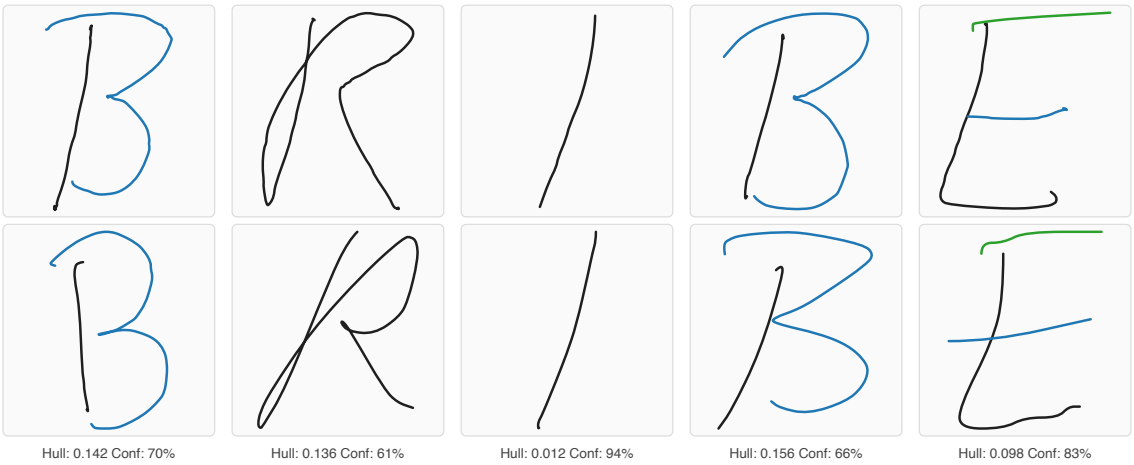

Figure 8.3: `BRIBE` → `BRlBE`. Superclass-equivalent recognition error: the capital `I` written as a plain vertical pen mark is matched against an equally plain vertical pen mark from the lowercase-l class.

Recognition: classifier fault on a clean input. Figure 8.4 shows a clean printed `R` with a vertical bar, a closed bowl, and a diagonal leg. A human reader does not hesitate. The matched sample makes the failure plain: the closest database sample for that position is a lowercase `n`. The two-stroke `R` written with a quick bowl shares its stroke count, its endpoint positions, and its overall arch shape with the n-class samples, and the projection coefficients land closer to those samples than to any `R`-class sample. The remaining five characters match correctly up to case. The exact-match metric still scores the whole word as wrong.

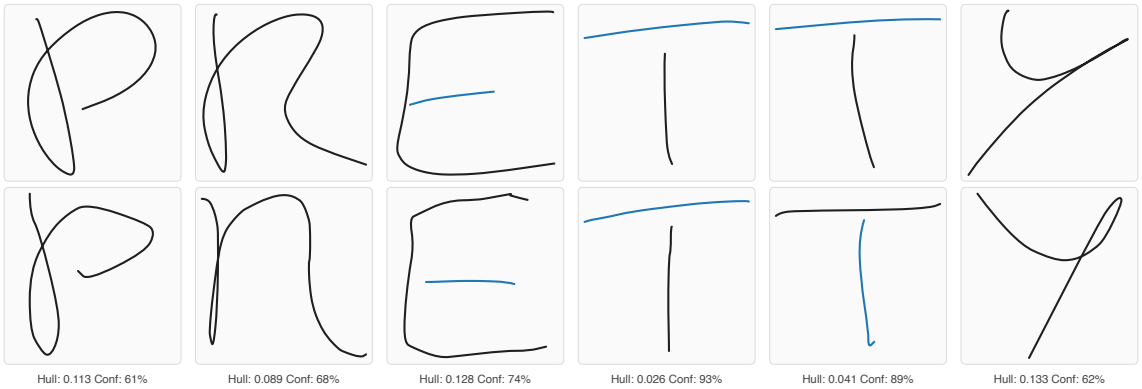

Figure 8.4: `PRETTY` → `pnETTY`. Classifier fault on a clean input: the matched sample for the second character is a lowercase `n` even though the written `R` has a clear bowl.

Classifier fault, single character drops a long word. Figure 8.5 is the same kind of single-character classifier fault inside a ten-character word. The writer drew a clean printed `A` at position zero; the matched sample is a lowercase `k`. The two diagonals of the written `A` together with the short crossbar match the geometry of a lowercase-k stem and kick more closely than they match the A-class samples. Nine of the ten matched samples are a strong visual match to what the writer drew. This is the most common failure pattern in the classifier-fault category: a long, correctly segmented word with one mis-classified character drops out of the exact-match count entirely.

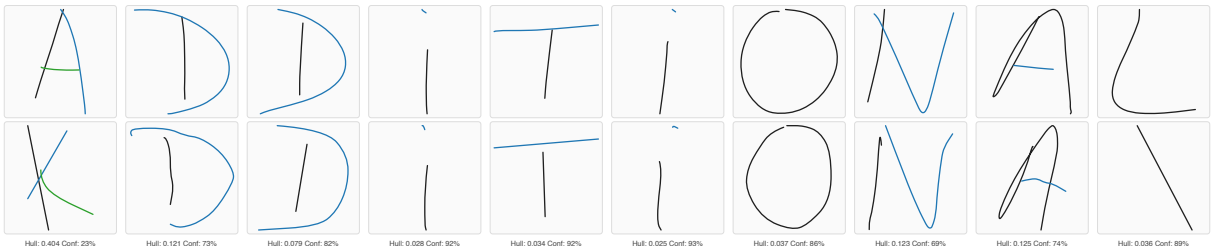

Figure 8.5: `ADDITIONAL` → `kDDiTioNAl`. A clean printed capital `A` returns lowercase `k`. Nine of the remaining matched samples agree with the written letters; the first does not.

Low-quality or difficult input. Figure 8.6 shows a three-character word with one wrong character. The pipeline matches the `A` to a lowercase `a` and the `I` to a capital `I`, both correct under case-insensitive matching. The failure sits on the third character. The writer drew a `D` with a small line across the top to mark the upper bar. A human reader picks up that line and reads the letter as `D` without trouble, but the pipeline sees a single pen-down stroke that closes back on itself and matches it against the lowercase-b class, whose bowl-and-stem geometry sits closer to the test stroke than any `D`-class sample after arc-length reparameterization. The wrong `D`-as-`b` match lands at 70% confidence with the lowest hull distance of the three columns. The classifier is more comfortable with this wrong answer than with either of the two correct ones. This example does not itself satisfy the more-than-half-wrong rule for the low-quality category, since only one of its three characters is misclassified; it is shown here to illustrate the kind of difficult input that gives rise to the category, not as a category member.

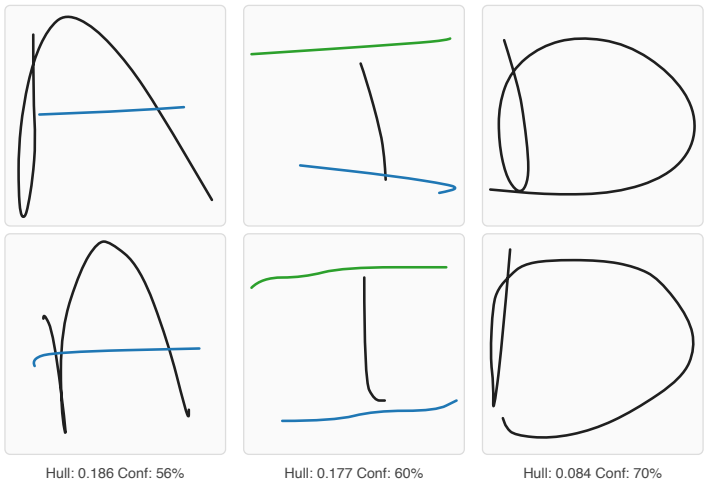

Figure 8.6: `AID` → `AIb`. Each column shows one character of the word: the writer's strokes on top, the closest database sample below, with hull distance and confidence reported underneath. The `I` matches a capital `I`; the `D` matches a lowercase `b` drawn with a single closed pen-down stroke.

8.2.3 Where the superclass gain comes from

Applying the superclass rule lifts the exact-word-match figure from 39.1% to 44.2%, an absolute gain of 5.1 percentage points. The gain is concentrated in a single group. Of the 154 additional words rescued by the rule, almost all of them differ from their labels only along the vertical-stroke group $\{\mathrm{I}, 1, \mathrm{i}, \mathrm{l}\}$, where the classifier's choice between a capital I, a lowercase l, and a lowercase i is a coin flip on a single vertical pen mark. `BRIBE` in Figure 8.3 is one such case; words like `All` (predicted `AlI`) and `Bill` (predicted `Blll`) follow the same pattern. They all fail the strict word-level metric and all come out correct when the vertical-stroke group is treated as one symbol.

The remaining superclass groups in Table 7.2 cover the rest of the lift, but each group on its own accounts for only a handful of words. The `0/O/o` group adds a few cases where a closed oval is labeled as one and predicted as another. `COFFEE` in Figure 8.7 is one such case: the second character is a single closed oval whose projection coefficients are indistinguishable from a digit-zero sample, and the predicted word `COFFEE` differs from the labeled word only at that position. The `5/S` group and the `U/V` group each contribute fewer than ten words. The size of the lift comes from one pervasive ambiguity rather than a scattered set of look-alike confusions.

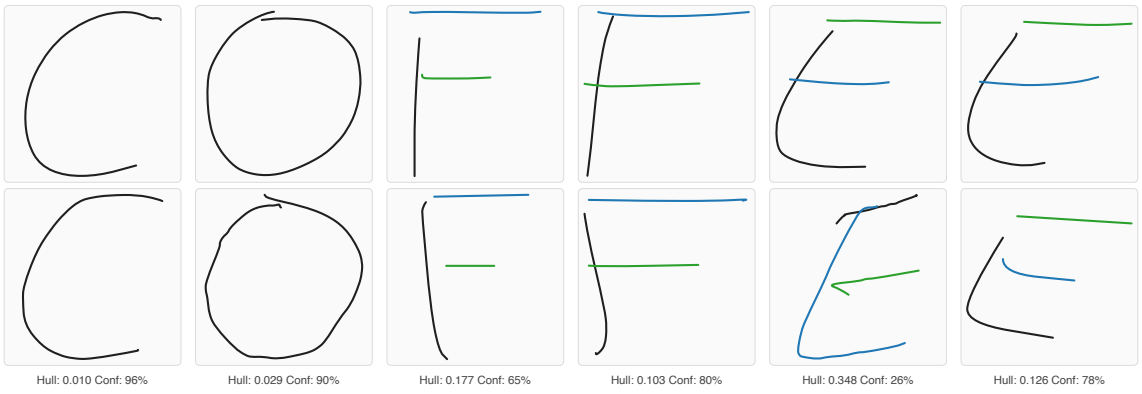

Figure 8.7: `COFFEE` → `COFFEE`. Superclass-equivalent recognition error on the `0/O` pair: the closed oval drawn as the second character is matched against a digit-zero sample with the same shape.

8.2.4 Frequent confusions inside the classifier-fault category

The 915 classifier-fault words are not 915 separate kinds of mistake. Most of them come from the same handful of letter pairs that the classifier mixes up over and over. Bahlmann and Burkhardt (2001) make the same observation in the HMM setting: a small number of character pairs account for most of the misclassifications [84]. Mazalov (2013) reports a similar concentration for the isolated-symbol classifier built on the same Legendre-Sobolev features used in this thesis [6]. Across the whole test set, 1,591 character positions disagree between predicted and labeled in a way that case-insensitive matching and the existing superclass groups do not absorb, and ten letter pairs alone account for 26.6% of those wrong positions. Table 8.3 lists them.

Labeled	Predicted	Count
n	m	111
l	e	66
R	n	47
r	v	36
A	k	33
n	x	33
s	r	25
s	O	25
r	n	25
g	y	23

Table 8.3: Top ten character-level confusion pairs after rejection, case, and superclass filtering.

Many of these pairs demonstrate how two letters can appear identical when written quickly. For example, an n with an additional wobble has a similar coefficient signature as an m with a double bump. An l or I with a little curl is very close to the D-class signatures, which are simply a vertical line with a curl. The main difference between t and f is the location of the horizontal crossbar, and the per-stroke shape feature does not capture that detail well. Many closed-loop letters, such as e, o, s, share gross shape with open loops such as c, G, r when the strokes are normalized to a length of 1. The classifier sees them as having the same shape, even though a human reader picks up the differences without

trouble. As stated above, Bahlmann (2006) states that another way to help distinguish between some of these loop types is by using both directional and coordinate data [68].

The fact that the classifier chose the best possible match of shapes, and therefore matched shapes in its own representation, does not mean the classifier was not functioning correctly. Mazalov and Watt (2012) concluded the same for the same type of features: there is a threshold after which improvements cannot be made unless the classifier uses information other than the per-stroke shape descriptor [12]. Therefore, a language model that understands what combinations of characters make valid English words or word-level context (for instance, the relative position of each character compared to their neighbours) would resolve these confusions without modifying the classifier [82, 29]. That direction is revisited in Chapter 9.

8.3 Discussion

Each of the four error categories needs a different kind of fix.

Trace-grouping (38.2% of wrong words). The stroke grouping produced the wrong number of character units before classification [3]. Figures 8.1 and 8.2 illustrate how the pipeline can fail in these two ways: by splitting one letter into two due to an internal gap (within character) or by joining two letters together using a very long stroke. The pipeline generates up to six different candidate segmentation possibilities for each input word, but it cannot select from those groups if the correct segmentation was not included as a candidate possibility. A larger segmentation search space is naturally the solution to this problem. Chapter 9 comes back to this direction.

Classifier fault (47.6% of wrong words). The strokes for the correct groupings were correct, but the classifier was unable to correctly determine the letter from those groups. This is because the per-stroke shape descriptors do not take into account the length (size) or the location of the current stroke relative to previous ones. Therefore, even if there is an attempt to create similar letters using the same pen-trace characteristics, they will appear to be identical. The confusions listed in Table 8.3 are primarily due to this type of comparison. A solution will have to include additional information that the current

shape descriptors do not utilize. Examples would be the height of the current character when it is compared to other characters in close proximity. These types of characteristics (ascenders/descenders, etc.) are common in cursive word recognition systems [66].

Superclass-equivalent (8.0% of wrong words). The scoring rule already handles these. They count as wrong under strict matching and correct under the superclass rule. The 5.1-percentage-point gap between the two word-accuracy figures in Table 8.1 is the size of that effect.

Low-quality or difficult input (6.1% of wrong words). The input is too rough for any per-character classifier to recover from. A fix sits upstream of the classifier altogether: a language model that knows English words [36], a way for the user to correct a wrong prediction, or a test set drawn from cleaner handwriting.

Two observations close this chapter. The character-level accuracy numbers in Table 8.1 (per-word: 75.1%, per-character: 83.8%) are well above the 39.1% exact word match. This gap follows from the fact that a single character error is enough to make an entire word wrong, while at the character level it counts as only one wrong position. If a word such as *long* is recognized as *loing* with one character wrong, the result counts as one character error at the character level but as a full word error at the word level. Word accuracy is therefore a stronger test of the entire process, while character accuracy is a fairer test of how well the per-stroke classifier works on its own. Classifier fault has the larger share of wrong words (47.6% to 38.2%), but trace-grouping errors are especially damaging because a wrong segmentation corrupts the whole word, while a single misclassified character inside a correctly segmented word still leaves the other characters intact [3]. Both stages are therefore reasonable directions for future improvement.

Chapter 9

Limitations and Future Work

9.1 Limitations

9.1.1 Limited segmentation search

Many of the Word Errors reported in Chapter 8 occurred because of an error made during the Trace Grouping process. As shown above, 38.2% of all incorrect word classifications resulted in either too many or too few character units being identified before the classification of each character. Even if the classifier was able to correctly identify each character unit, it still could have produced the wrong word due to poor segmentation. Segmentation-then-classification failures occur frequently throughout OCR systems, as noted by Casey and Lecolinet in their paper cataloguing segmentation failures across OCR systems [3].

The segmenter processes stroke pairs to rank possible segmentations and selects the top-ranking segmentation per word, with up to five alternative segmentations left available to be re-ranked by the classifier. For the classifier to recover from a poor initial segmentation of a word, the correct segmentation must be included among those six. There are primarily two segmentation types that cause the majority of errors. *Over-Segmentation*, where a single letter is split into two separate segments when the width of a within-character gap is greater than the merge threshold, as illustrated by the printed `H` in Figure 8.1. *Under-Segmentation*, where two letters are merged when they share a common stroke, such as a crossbar or a cursive ligature between them, as demonstrated by the `th` in Figure 8.2. In both of these scenarios, there is no way for the threshold rule to determine whether a within-character gap exists versus a between-character gap based solely upon the size of the gap itself.

A larger search space for segmentations would likely capture additional over and under-segmentation errors. The limited budget of six candidates used to select the *best* segmentation may not provide enough possibilities for capturing long words that can be segmented

into multiple different valid representations. The geometric threshold rules applied to score potential segmentations are also identical to those used by the grouper. A description of how a larger search space might be implemented to address this limitation is described in Section 9.2.1.

9.1.2 Per-stroke shape descriptor

The classification-failure class identified in Chapter 8 (47.6% of incorrect word choices) relates to the differences in what the per-stroke shape descriptor observes versus what it does not observe. Every stroke is represented by arc length and mapped onto a Legendre basis, which produces a constant-size coefficient vector that reflects the way the curve bends over its length. Shape descriptor features, such as the number of directional reversals, whether the curve intersects itself, whether it closes at some point, or where the sharp corners reside [54, 57], are not recorded.

The most persistent example of this occurs when the `A` and `k` letters are confused with each other, since they are drawn as a single stroke. In these instances, both letters occupy roughly the same area on the page: their starting/ending positions are very similar, they have nearly the same total height, and approximately the same center line position (zeroth Legendre coefficient). The distinguishing factor is where along the path the pen reverses direction, and how often it reverses. The Legendre basis represents that information using multiple mid-order coefficients rather than providing the classifier with a single value to evaluate.

Similarly, the gap in descriptors also contributes to the `s`-vs-`O` confusions. A single stroke `s` is an open curve with a relatively large end-point separation (0.786 in normalized coordinates) compared to a closed curve `O` with a smaller end-point separation (0.203) [54]. While a ratio of 3.9 × would normally provide a sufficient distinction between the two shapes, the Legendre coefficients fail to capture that distinction as a single numerical quantity. The endpoint separation can be derived from the coefficients. However, its magnitude varies across different handwriting styles for the same character. For example, a poorly formed `O` that fails to close has a larger separation than a cursive `s` that has a looped-back closure. Therefore, using this additional signal as input requires accounting for variations across different writing styles rather than applying a uniform additive penalty.

Sobolev distance weighting aids in this regard by increasing the weight of those mid-

order coefficients that represent curvature information. However, while helpful, there remains limited scope for improvement through this mechanism. As described by Teh and Chin (1988), regardless of how the coefficients may be weighted, orthogonal moments including Legendre representations will always be unable to distinguish between characters based solely upon their coefficient distributions within the tails [55].

9.1.3 Database composition

Recognition accuracy depends on what is in the reference database. The Latin-plus-digits database mixes digit and letter labels, and some digits are near-identical to letters in handwriting: `0` and `O`, `1` and `l`, `5` and `S`. Digits that are mapped to their letter equivalents during classification (Section 6.2) share hull space with the letter and help recognition. Digits that are not mapped (`2`, `3`, `4`, `6`, `7`, `8`) stay as separate competing labels and can pull test characters away from the correct letter.

Sample counts are also uneven. Common lowercase letters like `e` have dense class coverage (4,311 samples). Rare uppercase letters like `Q` have sparse coverage (396 samples). Adding uppercase samples from a broader source (Section 5.6) closed part of the gap, not all of it.

9.1.4 Loose or rushed handwriting

A number of misrecognized characters (6.1% in Chapter 8) result from handwriting errors where the per-character classifier cannot correct. Many writers have sloppy, hasty writing habits, which cause multiple letters to be written so as to resemble the same letter class. For example: when a writer writes their `o` closed too loosely it may appear as a `zero`, a `n` with some slop at the beginning may resemble an `x`, and a writer who always draws an additional loop at the bottom of their `n` will produce a sequence of strokes that resemble an `m`. Based on the data presented in Table 8.3 in Chapter 8, this is the single most frequent error in the test set. When a writer doesn't fully close off the bottom part of their `j` it appears as though they wrote a lower-case `l`. Each time the per-character classifier runs, it identifies the nearest neighbour in the database correctly, but since the original strokes were never anywhere near a similar shape for the correct character class, there was no way to recover the input.

No amount of weight tuning, basis switching, or database growth recovers an input that did not look like its label even before the classifier saw it. A language or context model can sometimes still rescue these by reading the surrounding word. A fix has to come from outside the per-character classifier: a language model that guesses the intended word from surrounding context [82, 29], an interactive correction step that lets the user disambiguate, or a test set drawn from cleaner handwriting.

9.1.5 Cursive and mathematical writing

The pipeline is designed to handle written Latin letters and standard handwritten letters. There are many types of input that fall outside of those categories. Cursive writing is one example. In cursive writing, the pen never lifts as it writes out a string of letters. Because the pen does not stop before a new character is written, the trace-grouping stage has no way to determine when one letter ends and another begins. Therefore, the trace-grouper uses a threshold to determine whether a group of strokes represents a single character or multiple. If an entire sentence is written with the pen moving continuously without lifting off the paper, then there is nothing for the grouper to see [16, 66]. Therefore, a cursive aware grouper will need to look at the path of the pen itself (the amount of curvature of each stroke, the speed of each stroke, etc.) rather than looking for breaks between strokes. Liwicki and Bunke used this approach and fit lines to the strokes of a person's hand while writing on a whiteboard [19].

The second type of input that falls outside of the categories handled by the per-stroke classifier is mathematical notation [5]. The per-stroke classifier knows how to recognize printed Latin letters and digits. However, it does not know anything about layout that creates expressions such as overbars, dot accents, subscripts and superscripts, fractions, square roots, or any number of layouts that turn a list of letters into a mathematical expression. Although the individual symbols may be recognized correctly during the classification process, the spatial arrangement that determines if a^b is different from ab, is not captured in the representation provided by the per-stroke classifier. Recognizing handwritten mathematical expressions requires a separate stage of processing above the existing pipeline that captures the layout information required to perform this task. The CROHME competition series is tracking advancements in solving the two-stage problem of recognizing symbols, followed by determining the structural arrangement of those symbols [39, 40].

9.1.6 Each character is classified on its own

The pipeline classifies one character position at a time. The decision for position i does not look at the decisions for positions $i-1$ and $i+1$, and it does not look at whether the resulting string forms a real word [82]. This is what produces predictions like `PNETTY` from a writer who clearly wrote `PRETTY`: the classifier picks the closest hull neighbour for the second character without checking whether `PNETTY` is a plausible English word. The architectural change for this sits outside the per-stroke classifier and is described in Section 9.2.4.

9.2 Future work

9.2.1 Wider segmentation search

The trace-grouping phase offers the largest potential area for improvement (Section 9.1). Currently, the classifier is given only a small fixed set of at most six candidate segmentations. A direct comparison against a single-candidate baseline is left to future work. Expanding the segmentation search could improve the results in two ways.

First, instead of using a fixed segmentation score evaluated on each candidate trace, the segmentation scores themselves could be learned. The current segmentation scores are calculated as a weighted sum of geometric features that include vertical gap, horizontal gap, bounding box overlap, and stroke count compatibility. These weights have been manually selected, but a learned segmentation score trained on labelled segmentations from a held-out validation set or a separately labelled development set would allow them to be adjusted to the true distribution of inter-character gaps. In fact, Hu and Zanibbi (2013) accomplish this exact process when they train their AdaBoost-based handwritten math symbol segmenter on shape context and geometric feature types [46]. This type of learning also maps directly into the geometric feature types used herein.

Second, the candidate budget should increase with word length. There are many more valid segmentations of a 26-character alphabet sequence than there are for a four-character word. Therefore, spending most of the candidate's budget on short words is wasteful. A length-aware candidate budget, or a prefix search over the classifier's output probability

used as a scoring function, would keep more possible segmentations open at longer word lengths, where most trace-grouping errors occur. As noted in Graves et al. (2006), CTC decoding performs a prefix search on per-frame classifier probabilities to select a correct sequence of labels from a much larger candidate pool than a fixed-size short list [24]. This would improve the 38.2% trace-grouping error share reported in Chapter 8.

9.2.2 Cutting cursive writing by stroke height

The current pipeline depends on identifying the separation between letters based on finding the space between the end of a letter and the beginning of another. In cursive writing, there are no spaces as the pen never lifts off. Therefore, it must be able to use some other indicator to identify where letters begin and end within the single continuous line.

A very simple indicator for this would be how high up the stroke was vertically. Typically, letter separations occur when the pen rises to either the top of the x-height or the bottom line, or when the horizontal movement of the pen ceases or changes direction [66, 19]. It could also look for all local minima and maxima of the y-coordinate, all local minima of horizontal velocity, and all *sharp turns* in the direction of the stroke. Then, that list of cut points would be passed to the segmentation-search algorithm described above. This would improve word-level accuracy on cursive inputs, which the current pipeline cannot segment at all.

9.2.3 Geometric shape features

The shape descriptor does not explicitly encode several geometric shape features (Section 9.1 above). Useful additions are the endpoint gap (open versus closed), the corner count (number of sharp direction changes above a threshold), and the winding number (how many times the curve wraps around a point). Zahn and Roskies (1972) develop their Fourier-descriptor framework on simple closed curves and note that it extends to nonclosed curves, so the open-versus-closed distinction is built into the representation rather than recovered from it. Feng, Kogan, and Krim (2010) build affine integral signatures that record global curve features in a similar spirit [54, 58]. Each of these is computable from the raw stroke data during normalization. The simple way to use them is to append the new values to the coefficient vector and re-train the convex-hull KNN on the wider feature

space. A better approach is to use them as a filter that rules out wrong classes before the KNN runs, since these features are closer to yes-or-no signals than smooth numbers, and treating them the same way as a polynomial coefficient hurts the distance more than it helps. This would improve the 47.6% classification-failure share attributed to the shape descriptor in Chapter 8, in particular the `A`-vs-`k` and `s`-vs-`O` confusions.

9.2.4 Language-model post-processing

The current pipeline classifies each character position independently. A language model or dictionary lookup applied after classification would resolve ambiguities by preferring character sequences that form real words. Zimmermann and Bunke (2004) report a word-error-rate drop from 34.6% to 29.6% on offline text recognition once an HMM classifier is paired with a word bigram language model [82], and Carbune et al. (2020) incorporate character and word language models into Google's production online recognizer [29]. If the pipeline predicts `hOuse` with the second character ambiguous between `o` and `O`, a language model would pick `o` because `house` is a word and `hOuse` is not. This extension would not change character-level accuracy on its own, but it should improve word-level accuracy by closing some of the gap between the 83.8% per-character figure and the 39.1% exact-word-match figure reported in Table 8.1.

Chapter 10

Conclusion

This thesis addresses online recognition of handwritten Latin words written as a stream of pen strokes. The hard part is not classifying a single character in isolation; it is deciding which strokes belong to which character before classification begins. A wrong cut at the segmentation stage corrupts every downstream prediction, and a per-character classifier on its own cannot recover from it.

The method keeps segmentation and classification loosely coupled. A probabilistic gap model produces up to six candidate stroke groupings per word rather than committing to a single segmentation. Each grouping is normalized into a common bounding box, reparameterized by arc length on $[-1, 1]$, and projected onto the degree-11 Legendre basis to give a fixed-length coefficient vector per stroke. Direction ambiguity is resolved by a geometric canonicalization rule that replaces an exponential search over stroke orientations with a constant-time choice. Each character is classified by a two-stage convex hull KNN that reduces the roughly three thousand reference classes to fifty before computing a simplex distance to a label-pooled neighbourhood. The final word is the candidate with the highest mean character confidence.

On the filtered UniPen word test set of 3,154 words, the pipeline reaches 39.1% exact word match, 83.8% character accuracy, and 44.2% / 85.7% under the superclass-aware rule. The high character-level accuracy shows that the per-stroke Legendre representation with hull-based KNN is often able to identify individual characters once a grouping is supplied. The much lower word-level accuracy shows how strongly word recognition depends on local errors: a single wrong character or a wrong segmentation can make the whole word incorrect.

The main lesson is that the orthogonal-polynomial representation makes deferred segmentation feasible: candidate groupings, normalization, feature extraction, and per-character scoring can all be expressed in the same fixed-format coefficient representation.

The limitation is the segmentation search itself. Six candidates are not enough on

cursive or tightly joined writing, and roughly thirty-eight percent of remaining word errors come from a stroke grouping the classifier cannot correct. A secondary limitation is the metric mismatch between the L^2 curve-distance weights used for class radii and the Sobolev weights used for hull distances, which leaves the confidence score as a heuristic rather than a calibrated probability.

The natural next step is to widen the segmentation search and rescore candidate words with a character-level language model. The pipeline already emits the top eight character alternatives at each position, so a lattice rescoring stage can be added without disturbing the upstream components.

References

[1] Réjean Plamondon and Sargur N. Srihari. Online and off-line handwriting recognition: A comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):63–84, 2000.

[2] Stephen M. Watt et al. Ink markup language (inkml). W3C Recommendation, 2011. World Wide Web Consortium (W3C), September 2011.

[3] Richard G. Casey and Eric Lecolinet. A survey of methods and strategies in character segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(7):690–706, 1996.

[4] Bruce W. Char and Stephen M. Watt. Representing and characterizing handwritten mathematical symbols through succinct functional approximation. In *9th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1198–1202, 2007.

[5] Richard Zanibbi and Dorothea Blostein. Recognition and retrieval of mathematical expressions. *International Journal on Document Analysis and Recognition (IJDAR)*, 15(4):331–357, 2012.

[6] Vadim Mazalov. *Advances in Manipulation and Recognition of Digital Ink*. PhD thesis, The University of Western Ontario, London, Ontario, Canada, 2013. Supervisor: Stephen M. Watt.

[7] Oleg Golubitsky and Stephen M. Watt. Distance-based classification of handwritten symbols. *International Journal on Document Analysis and Recognition (IJDAR)*, 13(2):133–146, 2010.

[8] Parisa Alvandi and Stephen M. Watt. The Legendre-Sobolev package and its applications in handwriting recognition. *Maple Transactions*, 2020.

[9] UNIPEN Consortium. Unipen data set of on-line (vectorial) handwriting - train_r01_v07, December 1999. Version: December 1999.

[10] Oleg Golubitsky and Stephen M. Watt. Online stroke modeling for handwriting recognition. In *Proceedings of the 18th Annual International Conference on Computer Science and Software Engineering (CASCON 2008)*, pages 72–80, Toronto, Canada, 2008. IBM Canada.

[11] Oleg Golubitsky and Stephen M. Watt. Online recognition of multi-stroke symbols with orthogonal series. In *Proceedings of the 10th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1265–1269, Barcelona, Spain, 2009. IEEE Computer Society.

[12] Vadim Mazalov and Stephen M. Watt. Improving isolated and in-context classification of handwritten characters. In *Proceedings of Document Recognition and Retrieval XIX (DRR 2012)*, volume 8297 of *SPIE Proceedings*, 2012.

[13] Vadim Mazalov and Stephen M. Watt. Recognition of relatively small handwritten characters or "size matters". In *Proceedings of the 13th International Conference on Frontiers in Handwriting Recognition (ICFHR 2012)*. IEEE, 2012.

[14] Parisa Alvandi and Stephen M. Watt. Handwriting feature extraction via Legendre-Sobolev matrix representation. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2019.

[15] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49, 1978.

[16] Charles C. Tappert. Cursive script recognition by elastic matching. *IBM Journal of Research and Development*, 26(6):765–771, 1982.

[17] Thad Starner, John Makhoul, Richard Schwartz, and George Chou. On-line cursive handwriting recognition using hidden Markov models and statistical grammars. In *Proceedings of the ARPA Human Language Technology Workshop*, pages 432–436, 1994.

[18] Markus Schenkel, Isabelle Guyon, and Don Henderson. On-line cursive script recognition using time-delay neural networks and hidden Markov models. *Machine Vision and Applications*, 8(4):215–223, 1995.

[19] Marcus Liwicki and Horst Bunke. HMM-based on-line recognition of handwritten whiteboard notes. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition (IWFHR)*, pages 595–599, 2006.

[20] Yoshua Bengio, Yann LeCun, Craig Nohl, and Chris Burges. LeRec: A NN/HMM hybrid for on-line handwriting recognition. *Neural Computation*, 7(5):1289–1303, 1995.

[21] Claus Bahlmann, Bernard Haasdonk, and Hans Burkhardt. On-line handwriting recognition with support vector machines: A kernel approach. In *Proceedings of the 8th International Workshop on Frontiers in Handwriting Recognition (IWFHR)*, pages 49–54, 2002.

[22] Claus Bahlmann and Hans Burkhardt. The writer independent online handwriting recognition system frog on hand and cluster generative statistical dynamic time warping. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(3):299–310, 2004.

[23] Vuokko Vuori, Matti Aksela, Ramūnas Girdziušas, Jorma Laaksonen, and Erkki Oja. On-line recognition of handwritten characters. In *Neural Networks Research Centre Biennial Report 2000–2002*, pages 105–115. Helsinki University of Technology, 2002.

[24] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pages 369–376, Pittsburgh, PA, USA, 2006.

[25] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, 2009.

[26] Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in Neural Information Processing Systems 21 (NIPS)*, pages 545–552, 2008.

[27] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385 of *Studies in Computational Intelligence*. Springer, 2012.

[28] Volkmar Frinken and Seiichi Uchida. Deep BLSTM neural networks for unconstrained continuous handwritten text recognition. In *13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 911–915, 2015.

[29] Victor Carbune, Pedro Gonnet, Thomas Deselaers, Henry A. Rowley, Alexander Daryin, Marcos Calvo, Li-Lun Wang, Daniel Keysers, Sandro Feuz, and Philippe Gervais. Fast multi-language LSTM-based online handwriting recognition. *International Journal on Document Analysis and Recognition (IJDAR)*, 23(2):89–102, 2020.

[30] Jianshu Zhang, Jun Du, and Lirong Dai. A GRU-based encoder-decoder approach with attention for online handwritten mathematical expression recognition. In *14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, pages 902–907, 2017.

[31] Jianshu Zhang, Jun Du, and Lirong Dai. Multi-scale attention with dense encoder for handwritten mathematical expression recognition. In *24th International Conference on Pattern Recognition (ICPR)*, pages 2245–2250, 2018.

[32] Jiaming Wang, Jun Du, Jianshu Zhang, and Zi-Rui Wang. Multi-modal attention network for handwritten mathematical expression recognition. In *15th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1181–1186, 2019.

[33] Wenqi Zhao, Liangcai Gao, Zuoyu Yan, Shuai Peng, Lin Du, and Ziyin Zhang. Handwritten mathematical expression recognition with bidirectionally trained transformer. In *16th International Conference on Document Analysis and Recognition (ICDAR)*, pages 570–584, 2021.

[34] Wenqi Zhao and Liangcai Gao. CoMER: Modeling coverage for transformer-based handwritten mathematical expression recognition. In *European Conference on Computer Vision (ECCV)*, pages 392–408, 2022.

[35] Michael Jungo, Beat Wolf, Andrii Maksai, Claudiu Musat, and Andreas Fischer. Character queries: A transformer-based approach to on-line handwritten character segmentation. In *17th International Conference on Document Analysis and Recognition (ICDAR)*, 2023.

[36] Anastasiia Fadeeva, Philippe Schlattner, Andrii Maksai, Mark Collier, Efi Kokiopoulou, Jesse Berent, and Claudiu Musat. Representing online handwriting for recognition in large vision-language models, 2024.

[37] Anh Duc Le and Masaki Nakagawa. A system for recognizing online handwritten mathematical expressions by using improved structural analysis. *International Journal on Document Analysis and Recognition (IJDAR)*, 19(4):305–319, 2016.

[38] Ernesto Tapia and Raúl Rojas. A survey on recognition of on-line handwritten mathematical notation. Technical Report B-07-01, Freie Universität Berlin, Fachbereich Mathematik und Informatik, 2007.

[39] Harold Mouchère, Richard Zanibbi, Utpal Garain, and Christian Viard-Gaudin. Advancing the state-of-the-art for handwritten math recognition: The CROHME competitions, 2011–2014. *International Journal on Document Analysis and Recognition (IJDAR)*, 19(2):173–189, 2016.

[40] Mahshad Mahdavi, Richard Zanibbi, Harold Mouchère, Christian Viard-Gaudin, and Utpal Garain. ICDAR 2019 CROHME + TFD: Competition on recognition of handwritten mathematical expressions and typeset formula detection. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*, pages 1533–1538, 2019.

[41] Joseph J. LaViola, Jr. and Robert C. Zeleznik. MathPad^2: A system for the creation and exploration of mathematical sketches. In *ACM SIGGRAPH 2004 Papers*, pages 432–440. ACM, 2004.

[42] Joseph J. LaViola, Jr. *Mathematical Sketching: A New Approach to Creating and Exploring Dynamic Illustrations*. PhD thesis, Brown University, 2005.

[43] Steve Smithies, Kevin Novins, and James Arvo. A handwriting-based equation editor. In *Proceedings of Graphics Interface*, pages 84–91, 1999.

[44] Ernesto Tapia and Raúl Rojas. Recognition of on-line handwritten mathematical expressions in the E-Chalk system. Technical Report B-03-13, Freie Universität Berlin, Fachbereich Mathematik und Informatik, 2003.

[45] Lei Hu and Richard Zanibbi. HMM-based recognition of online handwritten mathematical symbols using segmental K-means initialization and a modified pen-up/down feature. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*, pages 457–462, 2011.

[46] Lei Hu and Richard Zanibbi. Segmenting handwritten math symbols using AdaBoost and multi-scale shape context features. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*, pages 1180–1184, 2013.

[47] Lei Hu and Richard Zanibbi. Line-of-sight stroke graphs and Parzen shape context features for handwritten math formula representation and symbol segmentation. In *Proceedings of the International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 180–186, 2016.

[48] Lei Hu and Richard Zanibbi. MST-based visual parsing of online handwritten mathematical expressions. In *Proceedings of the International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 337–342, 2016.

[49] Kenny Davila, Stephanie Ludi, and Richard Zanibbi. Using off-line features and synthetic data for on-line handwritten math symbol recognition. In *Proceedings of the International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 323–328, 2014.

[50] Lei Hu. *Features and Algorithms for Visual Parsing of Handwritten Mathematical Expressions*. PhD thesis, Rochester Institute of Technology, 2016.

[51] Scott MacLean and George Labahn. A new approach for recognizing handwritten mathematics using relational grammars and fuzzy sets. *International Journal on Document Analysis and Recognition (IJDAR)*, 16(2):139–163, 2013.

[52] Scott MacLean and George Labahn. A Bayesian model for recognizing handwritten mathematical expressions. *Pattern Recognition*, 48(8):2433–2445, 2015.

[53] Ming-Kuei Hu. Visual pattern recognition by moment invariants. *IRE Transactions on Information Theory*, 8(2):179–187, 1962.

[54] Charles T. Zahn and Ralph Z. Roskies. Fourier descriptors for plane closed curves. *IEEE Transactions on Computers*, C-21(3):269–281, 1972.

[55] Cho-Huak Teh and Roland T. Chin. On image analysis by the methods of moments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):496–513, 1988.

[56] Stéphane Bres, Véronique Eglin, and Catherine Volpilhac-Auger. Evaluation of handwriting similarities using Hermite transform. In *Proceedings of the Tenth International Workshop on Frontiers in Handwriting Recognition (IWFHR)*, La Baule, France, 2006.

[57] Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522, 2002.

[58] Shuo Feng, Irina A. Kogan, and Hamid Krim. Classification of curves in 2D and 3D via affine integral signatures. *Acta Applicandae Mathematicae*, 109(3):903–937, 2010.

[59] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

[60] D. Randall Wilson and Tony R. Martinez. Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257–286, 2000.

[61] Pascal Vincent and Yoshua Bengio. K-Local hyperplane and convex distance nearest neighbor algorithms. In *Advances in Neural Information Processing Systems 14 (NIPS)*, pages 985–992. MIT Press, 2001.

[62] Isabelle Guyon, Lambert Schomaker, Réjean Plamondon, Mark Liberman, and Stan Janet. UNIPEN project of on-line data exchange and recognizer benchmarks. In *Proceedings of the 12th International Conference on Pattern Recognition (ICPR 1994)*, pages 29–33, Jerusalem, Israel, 1994. IAPR-IEEE.

[63] Louis Vuurpijl, Ralph Niels, Merijn van Erp, Lambert Schomaker, and Eugene Ratzlaff. Verifying the UNIPEN devset. In *Proceedings of the Ninth International Workshop on Frontiers in Handwriting Recognition (IWFHR)*, pages 586–591, 2004.

[64] Horst Bunke. Recognition of cursive Roman handwriting – past, present and future. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR)*, pages 448–459, 2003.

[65] R. Manmatha and Jamie L. Rothfeder. A scale space approach for automatically segmenting words from historical handwritten documents. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1212–1225, 2005.

[66] Alessandro Vinciarelli. A survey on off-line cursive word recognition. *Pattern Recognition*, 35(7):1433–1446, 2002.

[67] Charles C. Tappert, Ching Y. Suen, and Toru Wakahara. The state of the art in online handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787–808, 1990.

[68] Claus Bahlmann. Directional features in online handwriting recognition. *Pattern Recognition*, 39(1):115–125, 2006.

[69] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*. Springer, third edition, 2002.

[70] Eric Persoon and King-Sun Fu. Shape discrimination using Fourier descriptors. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-7(3):170–179, 1977.

[71] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[72] Parisa Alvandi and Stephen M. Watt. Real-time computation of Legendre-Sobolev approximations. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2018.

[73] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. Wiley-Interscience, 1973.

[74] Lambert R. B. Schomaker and Hans-Leo Teulings. Stroke- versus character-based recognition of on-line, connected cursive script. In J.-C. Simon and S. Impedovo, editors, *From Pixels to Features III*, pages 313–325. North-Holland, Amsterdam, 1992. Originally presented at IWFHR 1991.

[75] Matti Aksela. *Adaptive Combinations of Classifiers with Application to On-line Handwritten Character Recognition*. Doctoral thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, 2007.

[76] Oleg Golubitsky, Vadim Mazalov, and Stephen M. Watt. Orientation-independent recognition of handwritten characters with integral invariants. In *Proceedings of the Joint Conference of ASCM 2009 and MACIS 2009 (Asian Symposium of Computer Mathematics and Mathematical Aspects of Computer and Information Sciences)*, COE Lecture Note Vol. 22, Kyushu University, pages 252–261, Fukuoka, Japan, 2009.

[77] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[78] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[79] Evelyn Fix and Joseph L. Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. Technical Report Project 21-49-004, Report Number 4, USAF School of Aviation Medicine, Randolph Field, Texas, 1951.

[80] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[81] Oleg Golubitsky and Stephen M. Watt. Confidence measures in recognizing handwritten mathematical symbols. In *Conferences in Intelligent Computer Mathematics (CICM/MKM Workshops)*, 2009.

[82] Matthias Zimmermann and Horst Bunke. N-gram language models for offline handwritten text recognition. In *Proceedings of the 9th International Workshop on Frontiers in Handwriting Recognition (IWFHR)*, pages 294–299, 2004.

[83] Larry Yaeger, Richard Lyon, and Brandyn Webb. Effective training of a neural network character classifier for word recognition. In *Advances in Neural Information Processing Systems*, volume 9, pages 807–813. MIT Press, 1996.

[84] Claus Bahlmann and Hans Burkhardt. Measuring HMM similarity with the Bayes probability of error and its application to online handwriting recognition. In *Proceedings of the 6th International Conference on Document Analysis and Recognition (ICDAR)*, pages 406–411, 2001.

Algorithm pseudocode

This appendix gives a single-page pseudocode summary of the recognition pipeline developed in Chapter 6.

Algorithm A.1 Word recognition pipeline

Require: InkML word $\mathcal{T} = (t_1, \ldots, t_N)$, class database, sample database
Ensure: Best predicted word $\hat{w}$ with per-character confidences

$\{\mathcal{S}^{(c)}\}_{c=1}^{C} \leftarrow$ TraceGrouping($\mathcal{T}$) ▷ up to $C = 6$ candidates, Section 6.1
for each candidate segmentation $\mathcal{S}^{(c)}$ **do**
  $\hat{w}^{(c)} \leftarrow$ empty string
  for each character group $g \in \mathcal{S}^{(c)}$ **do**
    $g' \leftarrow$ Normalize(g) ▷ group bbox + arc length, Chapter 3
    $\mathbf{v} \leftarrow$ LegendreProject($g', d = 11$) ▷ 12 coefficients per axis per stroke, Chapter 4
    $\mathcal{C}_{\text{reduction}} \leftarrow$ ClassReduction($\mathbf{v}$, $N = 50$) ▷ Section 6.2.3
    for all labels ℓ with at least one class in $\mathcal{C}_{\text{reduction}}$ **do**
      $\mathcal{K}_\ell \leftarrow \text{KNN}(\mathbf{v}, \mathcal{C}_{\text{reduction}}, \ell, k = 7)$ ▷ label-pooled, Section 6.2.4
      $d_\ell \leftarrow$ HullDistance($\mathbf{v}$, $\mathcal{K}_\ell$) ▷ Section 6.3.1
      $\text{conf}_\ell \leftarrow \max(0, \min(1, 1 - d_\ell / r_\ell))$ ▷ per-label confidence, r_ℓ from contributing classes, Section 6.3.3
    end for
    $\ell^* \leftarrow \arg\max_\ell \text{conf}_\ell$ ▷ best label by maximum confidence
    $\text{conf}^* \leftarrow \text{conf}_{\ell^*}$
    if $\text{conf}^* < 0.05$ **then**
      $\ell^* \leftarrow$? ▷ rejection threshold
    end if
    append (ℓ^*, conf^*) to $\hat{w}^{(c)}$
  end for
  $\overline{\text{conf}}^{(c)} \leftarrow$ mean of per-character confidences in $\hat{w}^{(c)}$
end for
$c^* \leftarrow \arg\max_c \overline{\text{conf}}^{(c)}$ ▷ final word by mean character confidence, Section 6.4
$\hat{w} \leftarrow \hat{w}^{(c^*)}$
return $\hat{w}$ and its per-character confidences

The fixed parameters used in the primary evaluation (candidate budget $C = 6$, KNN $k = 7$, class-reduction count $N = 50$, Legendre degree $d = 11$, rejection threshold 0.05, class radius percentile 90) are collected in Table 7.3.