Handwritten Character Recognition with SQL

Sql is a Programming Language

Sescu Matei
University of Tübingen, Germany
matei.sescu@uni-tuebingen.de

ABSTRACT

Handwriting recognition converts handwritten text into digital form. While machine learning is commonly used for this, this paper shows how SQL can be used for step-by-step character recognition. Inspiration from the Graphical Input Language system was used to implement this. It processes single-stroke handwritten characters and uses predefined SQL rules to classify them. The system consists of a drawing canvas, a SQL database for feature extraction, and filters for matching input to known patterns. This method is simple to modify and offers an alternative to machine learning models.

1 INTRODUCTION

Handwriting recognition is a well-known problem in the field of human-computer interaction. Typically, this problem is solved using machine learning models. However, in this paper we explore a different approach. This is done in two parts:

(1) A Python-based canvas that prints out an SQL query to insert triplet - counter, x position and y position - into a table called `coords`.
(2) An SQL database that takes these coordinates and, through multiple processing steps and filters, returns the best matching character.

Different from handwriting recognition, as in strings of characters at once. In this way, the task can be managed in a Turing-complete programming language like `PL/pgSQL`[3]. This is done using the GRAIL method, which will be presented in the next section.

The motivation for this approach is using an alternative method to traditional AI/machine learning based handwriting recognition by taking advantage of SQL's querying, structuring, and filtering capabilities. This approach offers a straightforward way to recognize handwritten characters and avoids the complexity of machine learning models

2 THE GRAIL SYSTEM

The Graphical Input Language (GRAIL) system, developed by the RAND Corporation in the 1960s, aimed to offer a more natural way for people to interact with computers. At a time when few people knew how to use a keyboard, GRAIL provided an alternative for those who were not keyboard-literate. According to a 1966 RAND document, the goals of GRAIL were to "investigate methods by which a user may deal directly, naturally, and easily with [their] problem"[2].

Users interacted with the computer using a pen-like device on a tablet. Figure 1 shows what the system looked like. Users could draw on the tablet, and the system would recognize handwritten letters, numbers, punctuation, and geometric figures. The RAND tablet was a key component of the GRAIL system. The tablet had a 10.24" x 10.24" writing surface with a resolution of 100 lines per inch. The pen used with the tablet had a pressure-sensitive switch in the tip that notified the computer when the pen was pressed against the writing surface.

Instead of relying on trained models, the adapted GRAIL[1] process we used starts with a single-stroke drawing and proceeds through a step-by-step logical framework. It captures the stroke as (x, y) coordinates, then smooths and thins the data to reduce noise. The stroke is analyzed for directional changes and divided into 4 regions that span 90°: Right, Up, Left and Down. Then the code looks for sharp changes in direction and marks them as corners in a grid divided into 16 identical rectangles. We also store the grid position of the start and end points. The relationship between the sides of the rectangles creates a ratio that helps distinguish in some special cases that I'll discuss later.

3 WHY SQL AND PYTHON?

SQL is generally associated with data storage and retrieval rather than image or handwriting processing. However, SQL's structured query capabilities, combined with recursive common table expressions (CTEs), allow us to process and analyze handwriting systematically. This approach brings several advantages:

- Clear, rule-based structure for character recognition.
- Easy modification and expansion without retraining models.
- Transparency in feature extraction and decision-making.
- Efficient storage and retrieval of structured data for recognition tasks.

Python is used in this project to create a drawing interface and to convert the drawings into data that can be processed by the SQL backend (2). We chose it because it is easy to use and learn and it allows for quick prototyping and testing of ideas. Python also has a very useful library that helped me with creating the graphical user interface.

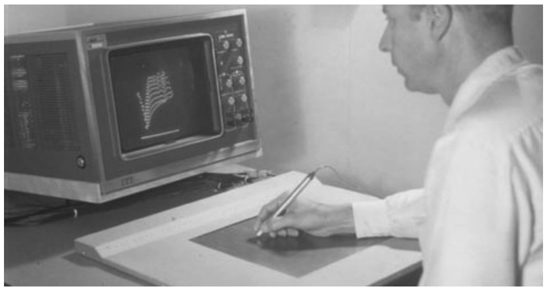

Figure 1: RAND system with tablet and pen

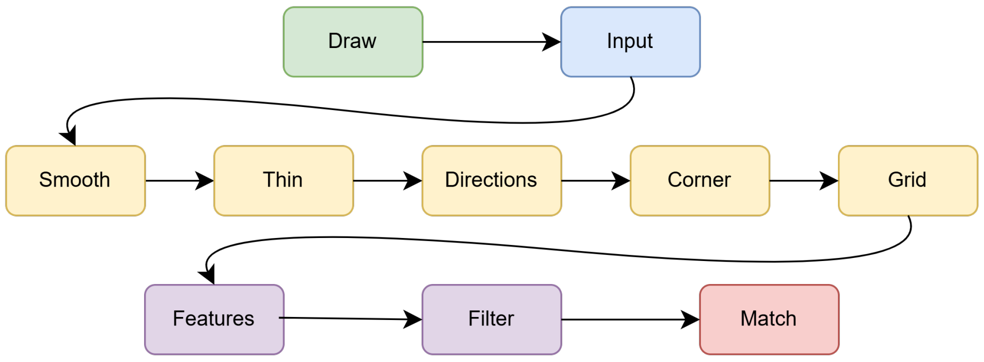

Figure 2: Project code steps overview

The canvas, shown in figure 3, is where the user draws a character. It's built with Python's Tkinter library, which is easy to use for creating simple GUIs. As the user draws, the program records a list of (x, y) coordinates representing the stroke. These coordinates are then converted into an SQL `INSERT` statement to populate a table called `coords`. This table is used by the SQL routines to analyze the drawing and recognize the character(3a).

4 HANDWRITING DATA PROCESSING

The raw data consists of a series of (x, y) coordinates that represent the drawn character. These points undergo several processing steps to ensure accurate recognition and meaningful feature extraction (2):

Smoothing reduces noise in the raw input by averaging nearby points (3b). Hand-drawn input often includes small fluctuations from shaking or inconsistencies, resulting in a jagged line. Smoothing creates a cleaner, more consistent line that better reflects the intended shape of the character.

With a smoother stroke, the data becomes more reliable for further analysis. This cleaner version improves the effectiveness of subsequent steps-such as thinning and directional assignment-by minimizing the impact of random variations and focusing on the overall shape of the character.

The smoothing can be adjusted by changing two numbers in the code, one for the x index and one for the y index. These 2 numbers need to be equal to ensure that the point slides along the imaginary line between itself and the previous point. In the snippet below, t is the current row and s is the previous row of the table with coordinates.

```
(0.70 * s.x + 0.30 * t.x) :: real AS x,
(0.70 * s.y + 0.30 * t.y) :: real AS y
```

Thinning removes extra dots that are too close together from the stroke (3c). Handwriting is naturally continuous, which often results in an excess of data points that can obscure the main structure of the character. Thinning condenses the data to retain only the essential points that define the outline of the character.

Reducing the number of points speeds up processing and highlights the key features of the stroke. This makes later analysis more efficient and helps the system focus on the most important aspects of the handwriting.

The thin recursive CTE processes the smooth data by keeping points that are sufficiently far apart. It starts with the first point, then for each subsequent point, it checks whether the distance to the previous point exceeds a specified threshold, in this case: 1. If the distance is less than the threshold, the point is excluded, otherwise it is included in the result. This step can be disabled by setting the threshold to 0.

```
thin(cnt, x, y) AS (
  SELECT s.cnt, s.x, s.y
  FROM smooth s
  WHERE s.cnt = 0

  UNION ALL

  SELECT s.cnt, s.x, s.y
  FROM smooth s
  LEFT JOIN smooth prev ON prev.cnt = s.cnt - 1
  WHERE prev.cnt IS NULL
     OR sqrt((s.x - prev.x)^2 + (s.y - prev.y)^2) > 1
)
```

Assigning directions involves breaking the stroke into segments and assigning each segment to one of four basic directions: Right, Up, Left, or Down. This step translates the continuous movement of the hand into a discrete sequence of directional movements, providing a simplified summary of the character's overall path.

The resulting sequence of directions serves as the basis for pattern matching against predefined templates in the SQL database. By converting complex curves into simple directional cues, this step facilitates a clearer comparison between the drawn character and stored patterns, improving recognition accuracy.

Corner detection identifies the points in the stroke where there is a sharp change in direction. These corners are crucial because they mark the boundaries between different segments of the character and capture its most distinctive features. By detecting these key transitions, the system gathers important structural information about the handwriting.

Corner detection helps to distinguish between characters that may have similar overall directional patterns, but differ in the number or placement of these critical points. This additional level of detail refines the matching process, contributing to a more robust and accurate recognition system.

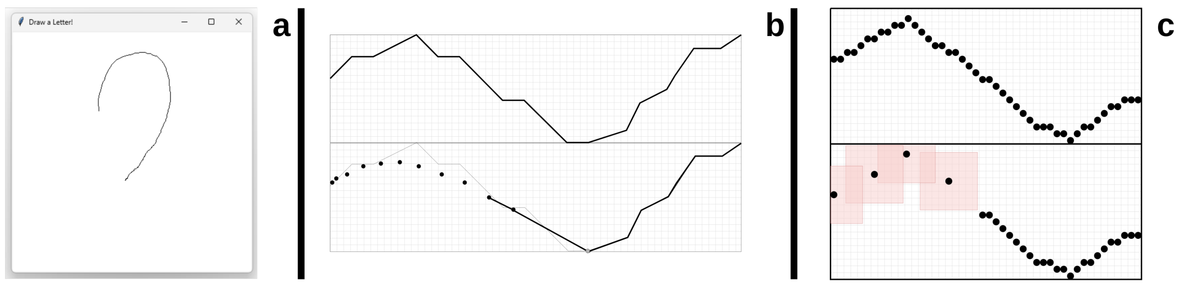

Figure 3: a. Python canvas window b. Smoothing process c. Thinning procces

Grid Mapping The Grid Mapping step normalizes the stroke by projecting it onto a fixed grid. It divides the drawing area into a predetermined number of cells and maps key features-such as start point, end point, and corners to specific grid locations (4). This normalization makes it easier to compare characters regardless of their original size or position.

Grid mapping standardizes the spatial layout, allowing for consistent comparison with stored templates. It ensures that variations in scale or placement don't affect the matching process, improving character recognition reliability.

The SQL function `grid_position` calculates a point's grid position based on cell size and minimum x and y values. It transforms the coordinates by subtracting the minimum values, then scales them by dividing by the cell dimensions. The scaled values are multiplied by 4 and floored to get discrete grid indices for x and y. The x index is adjusted by subtracting it from 3, and the y index is scaled by 4. The function combines these indices and ensures the result is non-negative using `GREATEST(0, ...)`. This process maps the coordinates to a finer grid, where the position is an integer value indicating a specific location.

```sql
CREATE OR REPLACE FUNCTION grid_position(
  width REAL, height REAL, xmin REAL, ymin REAL, x REAL,
     y REAL)
RETURNS INT AS $$
  SELECT GREATEST(0,
    (3 - (FLOOR(4 * (x - xmin) / (width + 1)) :: INT))
    + 4 * (FLOOR(4 * (y - ymin) / (height + 1)) :: INT)
  );
$$ LANGUAGE SQL;
```

5 SQL-BASED CHARACTER RECOGNITION

After the handwriting is pre-processed through steps such as smoothing, thinning and feature extraction, the resulting data is stored in an SQL database. The recognition process is then performed by systematically filtering out unlikely candidates until only the best match remains. This filtering is done in two main stages, with table inputs sourced mostly from [1]:

5.1 Possible Characters Table

The first stage uses a table that maps initial stroke motion sequences to potential characters. In handwriting, the early parts of a stroke carry significant information about its overall structure. For example, if the first few recorded directions are down, right, up, and left (denoted D, R, U, L), this pattern could be characteristic of characters such as O, X, or U.

By matching the initial directional sequence of the drawn stroke to those stored in the table, the system quickly reduces the universe of possible characters. This early filtering is critical because it narrows down the choices without having to analyze every detail of the stroke. Essentially, it acts as a broad categorization step - eliminating all characters whose initial stroke patterns don't match the observed data.

The `matched_possible_characters` CTE identifies potential matches by joining the `possible_characters` table with the `features` table based on the first few directions stored in the `directions` array. It compares the first four directions of the `f.directions` array with the `first_four_directions` of the pc table, using the `LEAST` function to handle cases where less than four directions are available. After identifying the potential candidates, it joins the result with the `criteria` table to filter out and return the candidate characters that match based on predefined criteria, such as shape or other characteristics.

```sql
matched_possible_characters AS (
  SELECT c.*
  FROM (
    SELECT pc.candidate_characters
          -- ,pc.first_four_directions
      FROM possible_characters pc
      JOIN features f
```

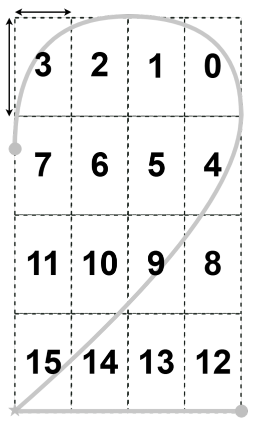

Figure 4: Grid elements

```
            ON f.directions[1:LEAST(4, array_length(f.directions, 1))] =
                                        pc.first_four_directions
    ) AS cc
    JOIN criteria c ON c.candidate_characters=cc.candidate_characters
)
```

5.2 Criteria Table

The criteria table refines the selection using additional stroke characteristics:

- **Start and end positions:** Differentiates characters by their starting and ending points.
- **Number and positions of corners:** Helps identify characters based on the number and placement of corners.
- **Aspect Ratio:** Compares height to width to distinguish between narrow or wide characters.
- **Alignment with saved templates:** Compares stroke details with pre-stored templates to ensure a closer match.

In practice, only the characters that meet all these criteria are passed on as final candidates for recognition. By integrating these multiple layers of filtering, the system can effectively balance between speed (via broad early filtering) and accuracy (through detailed later-stage analysis).

The `character_match` CTE finds the best match by comparing candidate features to a given stroke's properties. It selects key features from the `features` table, including `directions`, `start`, `end`, `corners`, `aspect`, and `center`. For each candidate, it calculates several scores: `corner_match_score` (matching corners), `aspect_score` (aspect ratio comparison), `start_end_match_score` (whether the start and end points match), and `center_proximity_score` (closeness of the center to a point).

The `UNNEST` function is used to expand an array or set into a table of individual rows. This is particularly useful when working with arrays of values, such as the corners of a stroke in this context.

These scores are combined into a `total_score`, which ranks the matches. The query filters based on `start`, `finish`, and aspect ratio, and sorts results by `total_score` (descending) and `aspect_score` (ascending). It then limits the result to the top match, ensuring the most accurate character is selected.

```
character_match AS (
    SELECT
        mc.character, f.directions, f.start, f.finish,
        f.corners, f.aspect, f.center,
    (
            SELECT COUNT(*)
            FROM UNNEST(f.corners) AS fc
            JOIN UNNEST(mc.corners) AS mc_c ON fc = mc_c
        ) AS corner_match_score,
        ABS(f.aspect - lower(mc.aspect_range)) +
    ABS(f.aspect - upper(mc.aspect_range)) AS aspect_score,
        CASE
            WHEN f.start = f.finish THEN 1
            ELSE 0
        END AS start_end_match_score,
        CASE
            WHEN sqrt((f.center[0] - s.x)^2 + (f.center[1] - s.y)^2) < 10
     THEN 1
            ELSE 0
        END AS center_proximity_score,
        (
            (
                SELECT COUNT(*)
                FROM UNNEST(f.corners) AS fc
                JOIN UNNEST(mc.corners) AS mc_c ON fc = mc_c
            ) +
            CASE
                WHEN f.start = f.finish THEN 1
                ELSE 0
            END +
            CASE
                WHEN sqrt((f.center[0] - s.x)^2
        + (f.center[1] - s.y)^2) < 10
        THEN 1
                ELSE 0
            END
        ) AS total_score,
        f.thin_coords
    FROM matched_possible_characters mc
    LEFT JOIN features f ON TRUE
    LEFT JOIN (
        SELECT x, y
        FROM smooth
        ORDER BY cnt DESC
        LIMIT 1
    ) s ON TRUE
    WHERE
        (mc.start IS NULL OR mc.start = f.start) AND
        (mc.finish IS NULL OR mc.finish = f.finish) AND
        (mc.aspect_range IS NULL OR f.aspect::NUMERIC <@ mc.aspect_range)
    ORDER BY
        total_score DESC,
        aspect_score ASC
    LIMIT 1 -- Best Match
)

SELECT *
FROM character_match;
```

6 RESULTS AND ANALYSIS

The system successfully identifies handwritten letters and numbers within its defined data set. The recognition process is fully deterministic and traceable, providing insight into how each feature contributes to the classification. In addition, step-by-step execution in SQL allows for incremental debugging and refinement, ensuring robust performance across different handwriting styles.

6.1 Case Study: Recognizing the Character 2

In an illustrative case, the system processes a drawn character intended to represent the digit **2** (6). First, the *possible characters* table filters candidates based on the first few stroke directions. In this case, the directional pattern of the drawn **2** closely matches several candidates, including **2** and **Z**. However, further analysis in the *criteria* table reveals that certain features of the drawn **2**, such as the start and end positions, the aspect ratio, and the positions of significant corners, are more consistent with the expected template of a **2**.

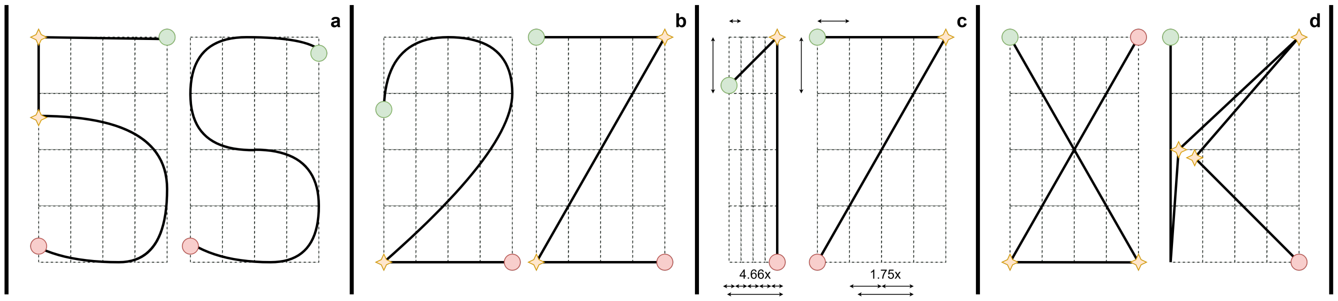

Figure 5: Notable cases of character differentiations

character	directions	start	finish	corners	aspect	center
2	{U,R,D,L,R}	7	12	{15}	1.5388119	-201,213

Figure 6: Resulted output for a drawn character 2

6.2 Differentiating Between Similar Characters

The recognition system is designed to handle characters with subtle differences through its layered filtering process:

5 vs. S Both **5** and **S** have a curvilinear structure, but the system distinguishes between them by analyzing corner placement and stroke flow (5a). A **5** typically has a more defined corner where the horizontal and vertical strokes meet, while a **S** tends to have a smoother, more continuous curve with less pronounced angles. These nuances, captured during the corner detection and later analysis stages, allow the system to accurately distinguish between the two.

2 vs. Z The system distinguishes between 2 and Z by analyzing stroke direction and corner placement (5b). The 2 character has a rounded bottom with an upward central stroke, while the Z character has sharp angles, especially where the horizontal and diagonal strokes meet. These structural differences are captured during corner detection and grid mapping, ensuring accurate classification.

1 vs. 7 Similarly, the system distinguishes between **1** and **7** by evaluating the aspect ratio and the endpoints of the strokes (5c). The **1** character generally has a tall and narrow structure with a straight stroke, while **7** is characterized by a diagonal stroke followed by a horizontal line. The positional data from the grid and the aspect ratio constraints ensure that even if the initial stroke directions are similar, the overall geometry will favor the correct classification.

6.3 Constraints of Single-Stroke Drawing

The single-stroke drawing approach, while effective for a lightweight system, introduces specific constraints:

- **Ambiguity in multi-stroke characters:** Characters such as **X** and **K** are conventionally drawn with multiple strokes. Forcing these characters into a single continuous stroke can obscure natural break points (5d), making it more difficult to capture their intended structure.
- **Loss of Detail:** In a single-stroke representation, the transition between what would naturally be separate strokes is smoothed, potentially obscuring critical features that aid in character recognition.
- **Increased Reliance on Geometric Constraints:** In the absence of clear stroke separations, the system must rely more heavily on geometric features, including aspect ratio and grid-based positional data, to distinguish between characters that share overlapping directional patterns.

Despite these challenges, the multi-layered SQL-based filtering process mitigates the impact of these constraints. The system's deterministic approach ensures that even with a single-stroke input, the extracted features are robust enough to support accurate character recognition across a variety of handwriting styles.

Overall, the combination of directional analysis and geometric constraints in a deterministic SQL framework provides both transparency and precision in the classification process, making it possible to trace exactly how each feature contributes to the final recognition decision.

7 FUTURE IMPROVEMENTS

While the current system achieves accurate character recognition using a structured SQL-based approach, there are ways to make this system even better. These improvements will make the system more reliable, easier to use, and more accurate. Some possible improvements include

Currently, the process of drawing a character and inserting its coordinate data into SQL is a manual operation. It would be a great improvement to create **automation between the Python canvas and the SQL recognition system**. This would allow users to immediately see how their drawn character is recognized, without having to copy and paste SQL insert statements. One idea is to run the SQL code inside Python, which is possible with the psycopg2 library. That way, the program that creates the interface would also be the one that executes the SQL code after each hit. It could be made into an exe and easily used by the owner of the database.

Despite the effectiveness of the **recognition pipeline**, certain characters remain inherently ambiguous. For example, the digit **5** and the letter **S**, or **1** and **7**, can have highly similar stroke patterns, making misclassification more likely. Improving edge case handling would significantly reduce misclassification rates and make the system more robust to different handwriting styles.

One of the key benefits of a deterministic SQL-based recognition system is its transparency. Unlike machine learning models, each step in the decision-making process can be traced and understood.

However, without a **visualization tool**, users currently have no insight into how the system is processing their handwriting. A graphical visualization could be implemented where:

- Users can see a step-by-step transformation of their drawing through smoothing, thinning, and direction extraction.
- The recognized character is displayed next to alternative candidates, with confidence levels for each.
- The system overlays the recognized directional sequences, corners, and grid positions on top of the user's input.

This feature would provide valuable insights and allow users to adjust their handwriting in real time for better recognition.

The recognition system currently relies on a fixed set of known character templates and predefined directional sequences. However, handwriting styles vary widely between individuals, and some edge cases may not be well represented in the current dataset. **Expanding the dataset** would require:

- Collect handwriting samples from a wide variety of users, ensuring coverage of different stroke variations.
- Store multiple valid representations for each character, increasing the flexibility of recognition.
- Implement a feedback system where misclassified input is logged and analyzed for improvement in future iterations.

By using a wider range of samples, the system can better understand different handwriting styles, improving accuracy and reliability.

8 CONCLUSION

This project shows that SQL, a language used for data management, can be used for handwriting recognition in a way that is easy to understand. By following a step-by-step process of preprocessing and filtering, we can achieve reliable character recognition without using complex machine learning models. This approach is particularly useful in situations where it is important to be transparent and able to change things. The step-by-step nature of SQL allows for organized, rule-based classification, demonstrating that even non-traditional tools can be used for pattern recognition tasks.

REFERENCES

[1] Jack Schaedler (2016). *Back to the Future of Handwriting Recognition*
[2] Gabriel F. Groner (1966). *Real-Time Recognition of Handprinted Text*
[3] The PostgreSQL Global Development Group (1996-2025) *PL/pgSQL documentation*