The entire code repository can be found at https://github.com/dgsaf/comp3007_assignment.

Contents

List of Figures

List of Tables

1 Problem Analysis

We are concerned with the detection and classification of two similar but distinct types of signs. The properties specific to each type of signage are discussed in subsection 1.1 and subsection 1.2. Here, we remark on the properties common to both of them, which are numerous and allow for a broadly unified approach to their detection and classification.
Each sign consists of a black background with white characters (digits and directional arrows) forming the foreground of the sign. The black background is rectangular in shape, and bounds the characters regions. Furthermore, the contrast between the black background and the white foreground is strong across all colour channels and in grayscale.
The digits are uniform in their construction, being printed in a monospace font with fixed height and allocated a fixed width; although each digit does not necessarily fully extend across its allocated width - as can be seen for the digit 1. Likewise, the directional arrows are also uniform in their construction but their dimensions are distinct from those of the digits - noticeably having a smaller height than the digits.
The digits do not overlap with each other (unless subject to camera artifacts/blurring) but are closely adjacent to each other, and regularly spaced. Each sign contains at least one sequence of three digits, which are ideally surrounded by a black sub-region of the sign; although for task 2, the gap between the leftmost digit and the edge of the sign can be very small.
In summary, the following properties of the signs are almost always observed, providing a robust foundation for their detection:

- The uniformly white colour of the characters;
- The uniformly black background of the sign;
- The strong contrast between the white characters and the black background across all red, green, and blue channels as well as in grayscale;
- The uniform height of the digits;
- The uniform spacing of the characters horizontally;
- The height of the digits exceeding the width;
- The uniform dimensions of the arrows;
- The presence of a chain of exactly three digits;
- A sizeable sub-region of the sign above and below the chain of digits, which is uniformly black.

However, the assumption of the above properties may be invalidated by any one of the (non-exhaustive) list of conditions:

- The presence of strong radial, or motion blurring which may cause character to overlap;
- The presence of strong shadowns, glare, or other variations in lighting conditions across the character;
- The presence of a foreign object, such as a sticker or a mark, on the sign (violating the assumptions on uniformity of layout) or across the characters (violating the assumptions on uniformity of characters).

It should also be noted that often there are other monospace white characters present near the sign - typically letters forming the name of the location marked by the sign. These characters can be distinguished from the digits (and arrows) by the fact that they almost always form chains of more than three characters, and presented on their own signs which almost never have a black background.

1.1 Task 1

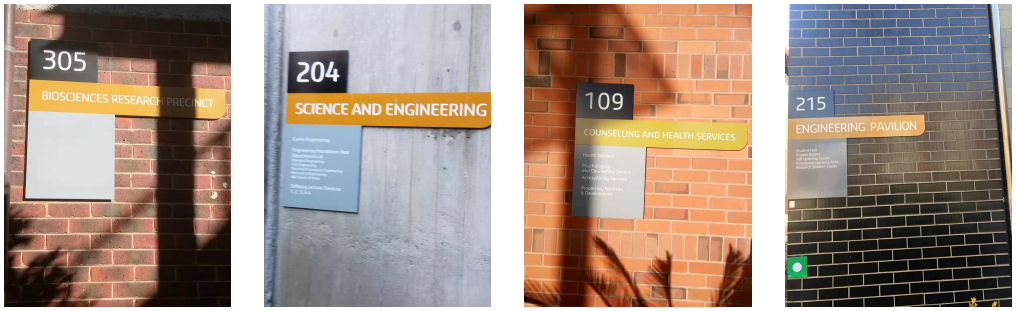
Further refinement of the problem analysis is possible for task 1. We are concerned with finding only one chain of three digits, and we are not concerned with directional arrows at all. Futhermore, there exists a sizeable gap between the chain of digits and the edge of the sign both horizontally and vertically.
Bricks and other objects which are of approximately uniform dimensions and uniformly spaced are regularly present in images for task 1 - hence, when exploiting the uniformity of the layout of the digits, care must be taken to avoid or handle the false detection of these other objects.
Examples of the images expected for task 1, which provide the justification for the assumptions made on the properties of the sign are shown in Figure 1. Examples of images which may be encountered for task 1, but which challenge the assumptions made on the properties of the sign are shown in Figure 2.

(a) `BS02.jpg` (b) `BS04.jpg` (c) `BS09.jpg` (d) `BS14.jpg`

Figure 1: Examples of training images for task 1, which exhibit the described distinguishing properties of the signs and digits.

1.2 Task 2

Similarly, further refinement of the problem analysis is possible for task 2. We are concerned with finding a variable number of chains of three digits, each of which has an associated directional arrow to their right. For each chain and their associated directional arrow, the geometric centres of the digits are arrow are collinear; hence we say that they form a line. Each line is vertically spaced (by a distance just larger than the height of the digits) and the set of digits and arrows from each line thus forms a grid-like layout.
A challenging aspect of these signs, compared to those for task 1, is that the gap between the leftmost digit on each line and the edge of the sign can be very small. In cases where an object, of similar colour/intensity to the white characters, is behind the sign but in a similar region of the



(a) `BS03.jpg` *(b)* `BS05.jpg` *(c)* `BS07.jpg` *(d)* `BS11.jpg`

Figure 2: Examples of training images for task 1, which may challenge the assumptions made on the signs and thus necessitate the development of a robust detection algorithm. Sharp shadows, motion blur and monochromatic periodic objects, which may interfere with digit detection, can be observed.

image to the leftmost digit of a line, the digit and the object may prove difficult to distinguish as separate regions.
Another difficulty is that the number of lines in any particular sign is not known beforehand. However, there is a benefit in having multiple lines per sign as it provides a mechanism, given at least one fully known line or even partial information from multiple lines, to recover an expected location for characters which fail to be detected initially.
Similar to task 1, the scene environment can be expected to be quite noisy, with periodically repeating objects such as brickwork present, as well as other objects with many sharp edges, such as plants and trees. It also seen that foreign objects, such as stickers and marks, may be present on the signs, and that other objects, such as plants, may occlude the sign itself.
Examples of the images expected for task 2, which provide the justification for the assumptions made on the properties of the sign are shown in Figure 3. Examples of images which may be encountered for task 2, but which challenge the assumptions made on the properties of the sign are shown in Figure 4.

(a) `DS06.jpg` *(b)* `DS12.jpg` *(c)* `DS13.jpg` *(d)* `DS18.jpg`

Figure 3: Examples of training images for task 2, which exhibit the described distinguishing properties of the signs, and their digits and arrows.

(a) `DS02.jpg` *(b)* `DS07.jpg` *(c)* `DS10.jpg` *(d)* `DS11.jpg`

Figure 4: Examples of training images for task 2, which may challenge the assumptions made on the signs and thus necessitate the development of a robust detection algorithm. Characters overlapping with similar regions outside the sign, foreign objects on the sign, variable lighting conditions can all be observed.

2 Implementation

2.1 Overview

We make use of the commonality of the properties of the signs that are to be detected in task 1 and task 2 by designing a detection algorithm which can be adapted with minor changes for each task. The specific detection algorithms for each task are discussed in subsection 2.5 and subsection 2.6. Furthermore, as the digits to be classified in each task are of the same style, we also design a classification algorithm which is suitable for both tasks. Naturally, a separate classifier will be needed for the directional arrows, however we use the k-Nearest Neighbours (k-NN) algorithm for classifying both the digits and the arrows. Hence, we discuss the problem of classification, for both types of characters, in a general manner in subsection 2.3.

2.2 Detection

To detect the characters (digits and arrows) we utilise the method of Maximally Stable Extremal Regions (MSER) - first developed by [Matas et al., 2004]. We present our formalism in a similar manner to [Matas et al., 2004]. In this method, a number of regions, which are extremal in terms of light intensity and stable in the sense that the region boundary changes little under a varying threshold, are detected in the grayscale image. We suggest that this technique is suitable for our purpose, as the characters are uniformly white and contrast well against the surrounding uniformly black bounding area of the signs. Furthermore, in most cases the characters are well separated from other characters and the edge of the sign (except notably for the leftmost digits in task 2). Hence, these characters should form extremal regions which are relatively stable under affine transformations, and when the image is underexposed or overexposed. Theoretically, this technique may be susceptible to extreme variations in lighting conditions, such as strong shadows or glare, which form discontinuous non-monotonic changes in light intensity. However, in testing, this technique has proved tolerant to moderate shadows and glare, so we do not expect this to be a major issue.

With suitably chosen parameters for the MSER algorithm, the regions of interest, corresponding to the characters on the sign, can be reliably detected - but must be distinguished from the other regions detected which are not of interest. Using known properties of the characters (such as dimension, aspect ratio, bounding box fill) we are able to refine the set of regions.

To detect the regions corresponding to the sequences of digits, we then examine the geometric relationships between the regions to determine which pairs of regions are: similar in dimension, and possibly collinear. This results in a directed graph of regions, with paths in this graph constituting possible lines of similar regions. For each region in this graph we filter their set of edges to leave only the edge connecting it to the closest adjacent region - ensuring all paths are non-overlapping. We then extract all possible paths, which we call chains, and analyse each chain by length and by contrast (in gray scale as well as in red, green, and blue channels) to order the chains by their likelihood of being a sequence of three digits.

For task 1, we detect only the most likely chain as our sequence of digits, and the problem of detection is complete. For task 2, we cluster these chains by their likelihood and select the most likely subset of the chains. If any of the sequences of digits have been only partially found (e.g. two out of three digits), we make use of the grid-like layout of the digits to locate the remaining digit. A bounding box is drawn around where the remaining digit is expected to be present, and a connected-components algorithm is employed to construct a region for this digit which is limited to

the scope of this bounding box. With all regions corresponding to the digit sequences now found, the collinearity of the digits is used to locate the region corresponding to the directional arrow associated with these digits. Thus for each task, the detected lines of digits (and directional arrows for task 2) are now located.

2.3 Classification by k-Nearest Neighbours

Prior to the classification stage, we note that we could use the collinearity of the lines, and the uniform height of the digits, to construct an affine, or even a perspective, transformation which rotates the line to be horizontal. However, it was observed in training and validation that the detection and classification algorithms performed well enough with out the need for a perspective transformation.
To address the problem of digit and directional arrow classification, we use the k-NN algorithm, comparing the regions expected to be digits and arrows with the training digits and arrows respectively. For each region, we define a binary image (with the dimensions of the bounding box of the region) which maps each point that is in the region to 1, and 0 otherwise. We then decompose the binary image into a set of bins (which are symmetric about the bounding box midpoint) and use the spatial occupancy of each bin as a feature vector for the region. We suggest that this approach is suitable, as the digits and directional arrows are of a uniform style and dimensions, and we note that this feature vector is scale-invariant. In testing with a suitably selected number of bins, this approach has proved to be reliably accurate despite its simplicity.

2.4 Mathematical Details

We introduce here various formal presentations of the concepts used in the detection and classification algorithms.

Image

We define an image I, with n channels and of width w and height h, to be a mapping

$$I : D \to S : (x, y) \mapsto I(x, y) = \big(I_1(x, y), \ldots, I_n(x, y)\big), \tag{1}$$

where the domain $D \subset \mathbb{N}^2$ is of the form $D = \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$, and S is the codomain. Constraints can be specified on S, but for our purposes it suffices to suppose that either $S \subseteq \{0, \ldots, 255\}^n$ or $S \subset \mathbb{R}^n$.

Thresholded Image

Suppose $I : D \to S \subset \mathbb{R}$ is a single-channel image. For each $t \in S$, we define the Boolean-valued image I_t to be of the form

$$I_t : D \to \mathbb{B} : (x, y) \mapsto \begin{cases} 0 & \text{if} \quad I(x, y) \leq t \\ 1 & \text{if} \quad I(x, y) > t \end{cases}, \tag{2}$$

and we say that I_t is a thresholded image.

Otsu Separation

Suppose that $I : D \to S \subset \mathbb{R}$ is a single channel image. Let $t \in S$ be the threshold which maximises the inter-class variance of the binary distribution $I_t(D)$, which is obtained via Otsu's method. The mean of each class $k \in \{0, 1\}$ is of the form

$$\mu_k(I) = \sum_{p \in D : I_t(p) = k} I(p). \tag{3}$$

We define the Otsu separation $\varsigma(I)$ of I to be the difference of these means; that is,

$$\varsigma(I) = \mu_1(I) - \mu_0(I). \tag{4}$$

Bounding Box

Suppose D is the domain of an image $I : D \to S$. For all $R \subseteq D$, we define the bounding box $B(R)$ of R to be

$$B(R) = [x_R, x_R + w_R] \times [y_R, y_R + h_R] \quad \text{such that} \quad R \subseteq B(R) \tag{5}$$

and where $w_R, h_R \in \mathbb{N}$ are minimal.

Connectedness

Suppose D is the domain of an image. An adjacency relation A on D is a Boolean-valued mapping

$$A : D \times D \to \mathbb{B} : (p, q) \mapsto A(p, q), \tag{6}$$

which indicates if the two points of the domain are considered to be adjacent. For any $p \in D$, we may define the neighbourhood $N(p)$ of p to be the set of all points which are adjacent to it; that is,

$$N(p) = \{q \in D \mid A(p, q)\}. \tag{7}$$

For any $p, q \in D$, we say that p and q are connected if there exists a finite sequence $(\rho_k)_{1 \leq k \leq n}$ in D such that

$$A(p, \rho_1) \wedge A(\rho_1, \rho_2) \wedge \cdots \wedge A(\rho_{n-1}, \rho_n) \wedge A(\rho_n, q) = 1. \tag{8}$$

In the case where the adjacency relation A is symmetric, then connectedness defines an equivalency relation; whence we may write, for any connected $p, q \in D$ that $p \sim q$.
Note that we are primarily concerned with the adjacency relations associated with the Von Neumann neighbourhood (4-connectivity)

$$N_4(p) = \{p + n \in D \mid n \in \{(0, 1), (1, 0), (0, -1), (-1, 0)\}\}, \tag{9}$$

and the Moore neighbourhood (8-connectivity)

$$N_8(p) = \{p + n \in D \mid n \in \{-1, 0, 1\} \times \{-1, 0, 1\} \setminus (0, 0)\}. \tag{10}$$

Region

Suppose D is the domain of an image $I : D \to S$ and let $A : D \times D \to \mathbb{B}$ be a symmetric adjacency relation on D. We say that $R \subseteq D$ is a region if every element of R is connected to every other element of R; that is,

$$p, q \in R \implies p \sim q. \tag{11}$$

We define the (inner) boundary ∂R of a region R to be subset of points of R which are also connected to at least one point not in R; that is,

$$\partial R = \{p \in R \mid \exists q \in D \setminus R : A(p, q)\}. \tag{12}$$

We define the outer boundary ΔR of a region R to be the set of points of D which do not belong to R but are adjacent to a point of R; that is,

$$\Delta R = \{p \in D \setminus R \mid \exists q \in R : A(p, q)\}. \tag{13}$$

Maximally Stable Extremal Region (MSER)

Suppose $I : D \to S \subset \mathbb{R}$ is a single-channel image, and suppose $A : D \times D \to \mathbb{B}$ is the (symmetric) adjacency relation associated with either the Von Neumann neighbourhood or the Moore neighbourhood. Suppose $R \subseteq D$ is a region. We say that R is a minimal region if for all $p \in R$ and $q \in \Delta R$ we have $I(p) < I(q)$; which is equivalently written as the requirement that

$$\max_{p \in R} I(p) < \min_{q \in \Delta R} I(q). \tag{14}$$

Similarly, we say that R is a maximal region if for all $p \in R$ and $q \in \Delta R$ we have $I(p) > I(q)$; which is equivalently written as the requirement that

$$\min_{p \in R} I(p) > \max_{q \in \Delta R} I(q). \tag{15}$$

We say that R is an extremal region if it is either a minimal or maximal region.
The formulation of extremal regions in terms of the minimal and maximal intensity values of the image permits the usage of thresholding. Suppose that R is an extremal region, and suppose that $t \in S$. Consider the thresholded region R_t, defined by

$$R_t = \{p \in R \mid I(p) < t\}, \tag{16}$$

which is itself an extremal region, and for which we have that

$$\max_{p \in R_t} I(p) < t. \tag{17}$$

We note that $R_t \subseteq R$ for all $t \in S$. We also note that when $t_1 \leq t_2$, we have that $R_{t_1} \subseteq R_{t_2}$; that is, the thresholded regions form an increasing (by set inclusion) sequence of subsets of R. For any increasing chain $t_1 < \ldots < t_n$ in S, we have

$$\emptyset \subseteq R_{t_1} \subseteq \cdots \subseteq R_{t_n} \subseteq R. \tag{18}$$

In the MSER approach, the stability of an extremal region R is measured by examining the change in the cardinality of R_t with the change in the threshold t. That is, for a particular threshold $t \in S$

and threshold step $\delta \in S$, such that $t - \delta, t + \delta \in S$, the rate of growth of the extremal region R is given by

$$G_\delta(R; t) = \frac{|R_{t+\delta} \setminus R_{t-\delta}|}{|R_t|}. \tag{19}$$

An extremal region R_{t_0} is then said to be maximally stable if $G_\delta(R; t)$ has a local minimum at $t = t_0$. Such a thresholded region experiences minimal change (in cardinality) when the threshold t_0 is increased/decreased by δ.

Spatial Occupancy

Suppose that D is the domain of an image $I : D \to S \subset \mathbb{R}$. Suppose that $R \subseteq D$ is a region and that $B(R)$ it its corresponding bounded box, with width w_R and height h_R. We now describe how we partition $B(R)$ into a number of rectangular bins which are symmetric about the midpoint of $B(R)$. Let $k_x, k_y \in \mathbb{N}$ be the number of x and y bins respectively. To simplify the following expressions, we employ the coordinate transformation $(x, y) \mapsto (x', y') = (x - x_R, y - y_R)$, for which $B(R) = [0, w_R] \times [0, h_R]$. We partition the x' component of $B(R)$, $[0, w_R]$, by the chain

$$0 = x_0 < x_1 < \cdots < x_{k_x - 1} < x_{k_x} = w_R, \tag{20}$$

where

$$s_x = \left\lceil \frac{w_R}{k_x} \right\rceil, \quad c_x = \left\lfloor \frac{w_R - (k_x - 2)s_x}{2} \right\rfloor, \quad x_n = \begin{cases} 0 & \text{if} \quad n = 0 \\ c_x + (n-1)s_x & \text{if} \quad 1 \leq n \leq k_x - 1 \\ w_R & \text{if} \quad n = k_x \end{cases}. \tag{21}$$

We similarly partition the y' component of $B(R)$ as above, replacing w_R by h_R, and variables subscripted by x with y. Each partition (or bin) of $B(R)$ is then of the form

$$B_{i,j} = [x_i, x_{i+1}] \times [y_j, y_{j+1}] \quad \text{for all} \quad 0 \leq i \leq k_x - 1, \quad 0 \leq j \leq k_y - 1. \tag{22}$$

We note that while $B_{i,j}$ is the Cartesian product of real intervals, $B_{i,j} \cap \mathbb{N}^2$ is the set of pairs of natural numbers which lie within these intervals. Using this partition of $B(R)$, we now define the spatial occupancy $v(R) = (v_{i,j}(R)) \in \mathbb{R}^{k_x \times k_y}$ of R as

$$v_{i,j}(R) = \frac{|R \cap (B_{i,j} \cap \mathbb{N}^2)|}{|B_{i,j} \cap \mathbb{N}^2|} \quad \text{for all} \quad 0 \leq i \leq k_x - 1, \quad 0 \leq j \leq k_y - 1; \tag{23}$$

that is, the fraction of discrete points in $B_{i,j}$ that are also in R. We note that $v_{i,j}(R) \in [0, 1]$ for all $0 \leq i \leq k_x - 1$, $0 \leq j \leq k_y - 1$.

2.5 Task 1

We describe the detection and classification algorithms for task 1 in detail.

1. We begin with the supplied colour image, $I_C : D \to \{0, \ldots, 255\}^3$, for which we are to detect a sign with three digits. We denote the red, green, and blue channels of this image by $I_R, I_G, I_B : D \to \{0, \ldots, 255\}$.

2. We create a grayscale transformation of this image, $I : D \to \{0, \ldots, 255\}$.

3. We use the MSER method, with the following parameters:

   - minimum region size of 45;
   - maximum region size of 2000;
   - threshold step, $\delta = 20$.

   Applying the MSER algorithm with these parameters to the grayscale image I yields a set of maximally stable extremal regions $\mathcal{R} = \{R_1, \ldots, R_n\}$.

4. The MSER algorithm can yield nested regions, or strongly overlapping regions, which interfere with the construction of chains of regions. Hence, we order the regions by decreasing area and filter out any regions which strongly overlap with a larger region. That is, we order $\mathcal{R}$ such that for all $R_i, R_j \in \mathcal{R}$ we have $|R_i| \geq |R_j|$ if $i \leq j$. Then, for each $R_i \in \mathcal{R}$, for all $R_j \in \mathcal{R}$ with $j \geq i$, we remove R_j from $\mathcal{R}$ if

   $$\frac{|R_j \cap R_i|}{|R_j|} \geq \lambda.$$

   In testing, we have found $\lambda = 0.8$ to be suitable (although this could probably be much stricter).

5. We then make use of the regularity of the aspect ratio of the bounding boxes of the digits to further filter $\mathcal{R}$. This step is effective at removing any regions corresponding to bricks (which tend to be wider than they are tall) which can be numerous, and thus impact the performance of the algorithm. For each region $R_i \in \mathcal{R}$ with corresponding bounding box $B(R_i)$, with aspect ratio $a_i = w_{R_i}/h_{R_i}$, we remove R_i from $\mathcal{R}$ if $a_i \notin [a_{\min}, a_{\max}]$. In testing, we have found $a_{\min} = 1:3$ and $a_{\max} = 1:1.2$ to be suitable.

6. A consequence of the MSER algorithm is that it detects the interior black holes of digits - such as those in 0, 6, 8, 9 - as extremal regions in addition to the white digits regions. These regions can also interfere with the construction of chains of regions, and so we filter them out. We make use of the fact that the bounding box of a region, that is contained within another region, will be contained in the bounding box of that region also, and that these interior regions we wish to remove are simply connected (they have no holes of their own). However, we wish to remove only interior regions which are closely packed into another region - and avoid accidentally removing say a character region which may possibly be inside a region corresponding to the sign. Hence, for each region $R_i \in \mathcal{R}$ we remove any region $R_j \in \mathcal{R}$ from $\mathcal{R}$ if

   $$B(R_j) \subseteq B(R_i) \quad \text{and} \quad \max_{q \in \partial R_j} \left(\min_{p \in \partial R_i} \text{dist}(p, q) \right) \leq \mu.$$

   That is, we only remove R_j if the maximum distance between the boundaries of R_i and R_j is less than μ. In testing, we have found $\mu = 10$ to be suitable (however, this should probably be modified to be scale invariant).

7. We then make use of the regularity of the fill of the digits regions to further filter $\mathcal{R}$, by removing any regions which fill their bounding box above a certain point. This step is effective at removing any vertically oriented rectangles which might otherwise be possibly mistaken for

digits. That is, for each region $R_i \in \mathcal{R}$, with corresponding bounding box $B(R_i)$, we remove R_i from $\mathcal{R}$ if

$$\frac{|R_i|}{|B(R_i) \cap \mathbb{N}^2|} \geq \xi.$$

In testing, we have found $\xi = 0.85$ to be suitable.

8. Having now filtered the regions quite heavily, we consider the dimensional similarity and geometric adjacency between regions to link sequences of similar, collinear regions into chains. We make use of the fact that the most reliable geometric property of the digits, which is least varying with perspective transformations, is the height of the bounding boxes of the digits. For all regions $R_i, R_j \in \mathcal{R}$:

   - We consider their height to be similar if

   $$\left|\frac{h_{R_i} - h_{R_j}}{h_{R_i}}\right| \leq \zeta_h \quad \text{and} \quad \left|\frac{h_{R_i} - h_{R_j}}{h_{R_j}}\right| \leq \zeta_h.$$

   In testing, we have found that $\zeta_h = 0.2$ to be suitable.

   - We consider their y placement to be similar if

   $$\left|\frac{y_{R_i} - y_{R_j}}{h_{R_i}}\right| \leq \zeta_y \quad \text{and} \quad \left|\frac{y_{R_i} - y_{R_j}}{h_{R_j}}\right| \leq \zeta_y.$$

   In testing, we have found that $\zeta_y = 0.5$ to be suitable.

   - We consider them to be x adjacent if

   $$\left|\frac{x_{R_i} - x_{R_j}}{h_{R_i}}\right| \leq \zeta_x \quad \text{and} \quad \left|\frac{x_{R_i} - x_{R_j}}{h_{R_j}}\right| \leq \zeta_x,$$

   and if the overlap of their boxes satisfies

   $$\frac{|B(R_i) \cap B(R_j)|}{|B(R_i)|} \leq \zeta_B \quad \text{and} \quad \frac{|B(R_i) \cap B(R_j)|}{|B(R_j)|} \leq \zeta_B.$$

   In testing, we have found that $\zeta_x = 1.0$ and $\zeta_B = 0.25$ to be suitable.

   If R_i and R_j are found to satisfy the above properties, we say that they are linked. We first find all links between all regions in $\mathcal{R}$, and then we filter these links - by minimal region-to-region distance - to ensure that each region is only linked to at most one region to its left and to at most one region on its right. As a result, all paths extracted from the graph formed by these regions and their edges will be non-overlapping. We extract all paths (of at least 2 regions) and refer to these paths as chains; that is, the chains $C_i \in \mathcal{C}$ are of the form $C_i = \{R_{i_1}, R_{i_2}, \ldots, R_{i_{n_i}}\}$.

9. It now remains to select the chain corresponding to be the sequence of three digits. While it would be possible to achieve this by classifying every digit in every chain and selecting the chain with the smallest k-NN distances, an alternatively simpler and seemingly reliable method has been employed instead. The white digits on the black background are likely to be the only predominantly black and white region in the image that we are considering. Hence, we need only select the most monochromatic chain, which has the sharpest contrast. For each chain $C_i = \{R_{i_1}, R_{i_2}, R_{i_3}\}$ of exactly three regions, we perform the following:

- From the bounding boxes of each region in the chain $B(R_{i_j})$ we construct the bounding box of the chain $B(C_i)$.
- For each color channel I_R, I_G, I_B and in grayscale I, the Otsu separation of the image, restricted to this bounding box, is calculated, and the cross-channel Otsu separation of this chain is taken to be the minimum of these values; that is, with slight abuse of notation,

$$\varsigma(C_i) = \min\{\varsigma(J\mid_{B(C_i)}) \mid J \in \{I_R, I_G, I_B, I\}\}.$$

  For a black and white region, the Otsu separation will be similar across each color channel and in grayscale, whereas for a coloured region it is likely that at least one will be smaller than the rest.

We then take the chain C_i, for which $\varsigma(C_i)$ is maximal, to be the chain associated with the sequence of three monochromatic digits. While this approach is not guaranteed to locate the sequence of digits, as opposed to another arbitrary chain of three equally spaced monochromatic regions, it is has proven successful across all training and validation images. Extremely unfortunate circumstances would be required to result in an arbitrary non-digit chain being selected ahead.

10. For each of the three regions in the chain of digits, we calculate its spatial occupancy feature vector $v(R_{i_j}) \in \mathbb{R}^{5\times 7}$ with 5 x-bins and 7 y-bins. These number of bins have been chosen to allow for a compact, low-dimensional feature vector which:

    - Can distinguish between digits with similar outlines but with a different number of holes, such as the digits 0 and 8;
    - Can distinguish between digits which are rotationally similar, such as the digits 6 and 9;
    - Can distinguish between digits which may possess similar sub-structures, such as the digits 0 and 6, or 1 and 7.

    These digits are then classified using a k-Nearest Neighbour algorithm, which is trained on the provided training digit images.

This detection algorithm is demonstrated in Figure 5.

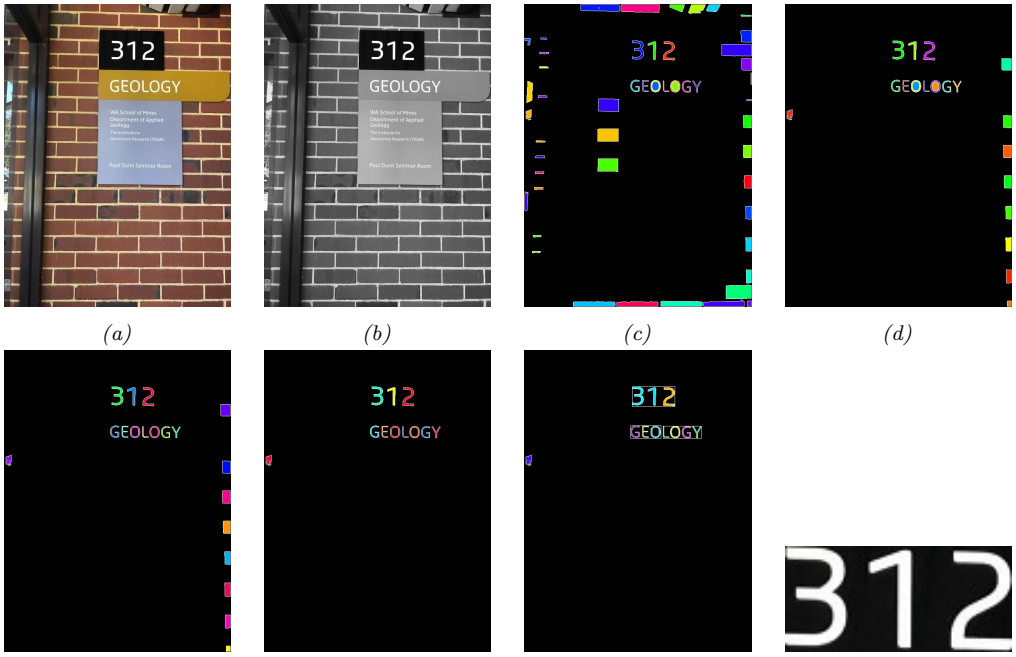


Figure 5: Demonstration of the detection algorithm for task 1, applied to `task1/val06.jpg`*: a - Original image; b - Coverted to grayscale; c - Regions detected by MSER; d - Overlapping regions filtered, and regions filtered by aspect ratio; e - Interior hole regions filtered; f - Regions filtered by fill; g - Chains detected; h - Most monochromatic chain (of three regions) selected (by Otsu separation).*

2.6 Task 2

We describe the detection and classification algorithm for task 2 in detail. The first stage of detection is essentially similar to that for task 1, as described in subsection 2.5, up to item 9 where the most likely chain is selected. The notable differences are that

- In the MSER method, we set a minimum area size of 25 rather than 45 as for task 1, due to the smaller size of the digits being detected;
- When filtering regions by aspect ratio, we set $a_{\max} = 1 : 0.75$ rather than $a_{\max} = 1 : 1.2$, so as to not exclude the arrow regions.
- No use is made of Otsu separation in task 2, rather likely chains are selected by considering their alignment with other chains.

We now describe the detection algorithm for task 2 assuming that, similar to what was done for task 1, we have obtained set of regions $\mathcal{R}$, and a set of chains $\mathcal{C}$ of similar and adjacent regions, with each chain containing at most three regions. We assume that if a chain, corresponding to a digit sequence, contains only two regions then it is missing its leftmost digit (due to the leftmost digit often being very close to the edge of the sign). Continuing from item 9:

10. We consider the alignment of the chains to partition the set of chains $\mathcal{C}$ into disjoint graphs. That is, for all $C_i, C_j \in \mathcal{C}$, we consider $C_i = \{R_{i_1}, \ldots, R_{i_{n_i}}\}$ and $C_j = \{R_{j_1}, \ldots, R_{j_{n_j}}\}$ to have an edge if:

    - The two rightmost regions of each chain are of similar height, in the sense defined in item 8; and
    - For each chain a bounding box is constructed around the two rightmost digits,

    $$B_i = B(R_{i_{n_i}} \cup R_{i_{n_i-1}}) \quad \text{and} \quad B_j = B(R_{j_{n_j}} \cup R_{j_{n_j-1}}).$$

    The boxes are considered to be aligned vertically if the x-difference between the midpoints of each box is smaller than the width of each box.

    This partitions $\mathcal{C}$ into a set of disjoint graphs, and we take the vertices of the largest, in terms of cardinality, graph $\mathcal{C}_A$ as the set of aligned chains - assumed to be set of (partial) digit sequences.

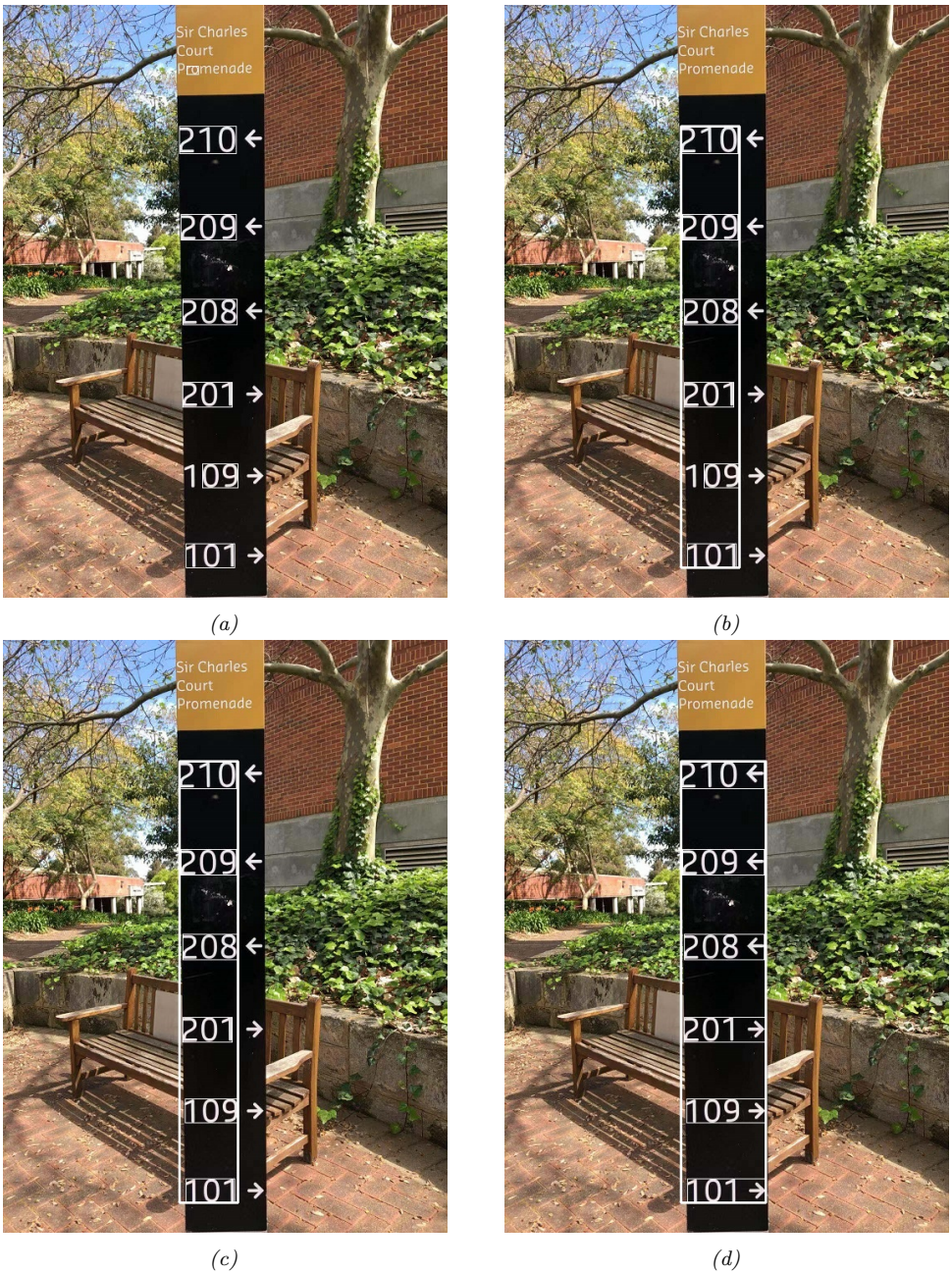
11. For each chain $C \in \mathcal{C}_A$ containing only two regions, we (re-)construct a region for its missing digit. We assume that its leftmost digit is missing, and we write the chain in the form $C = \{R_2, R_3\}$. We synthetically construct a region R_1, which allows replacing the chain C with $C' = \{R_1, R_2, R_3\}$. We construct a box B_1 to the left of R_2, with the top left corner being predicted from the spacing between the top left corners of R_2 and R_3, the height being the maximum of h_{R_2} and h_{R_3}, and its width extending the box to just before $B(R_2)$. The restriction of the grayscale image to this box $I\,|_{B_1}$ is subject to Otsu thresholding, and the connected components of the resulting binary image are determined. We expect B_1 to cover the leftmost digit, allowing the identification of the region corresponding to the digit, without letting the region extend outside B_1. We take R_1 to be the region formed from the largest connected component found, and replace C with C' in $\mathcal{C}_A$. Performing this for all incomplement chains in $\mathcal{C}_A$ yields a set of aligned chains, all of which consist of three regions.

12. We now find the directional arrow associated with each chain in the set of aligned chains. For each chain $C = \{R_1, R_2, R_3\} \in \mathcal{C}_A$, we construct a box B_4 to its right, which we expect to overlap maximally with a region $R_4 \in \mathcal{R}$ corresponding to the associated directional arrow. We construct B_4 by first constructing the bounding box of the two rightmost digits $B(R_2 \cup R_3)$ and shifting it rightwards, so that the x difference between the top left corner of B_4 and the top right corner of $B(R_2 \cup R_3)$ is 1. We then take R_4 to be the region $R_i \in \mathcal{R}$ for which the overlap of its corresponding bounding box and B_4 is maximal.

13. We have thus formed a set of aligned chains $\mathcal{C}_A$, each of which contain three regions corresponding to three digits $C_i = \{R_{i_1}, R_{i_2}, R_{i_3}\}$ and have an associated region R_{i_4} corresponding to a directional arrow. For each chain $C_i \in \mathcal{C}_A$ we perform the following:

    - For each of the three digit regions in the chain of digits, we calculate its spatial occupancy feature vector $v(R_{i_j}) \in \mathbb{R}^{3\times5}$ with 3 x-bins and 5 y-bins, for $j = 1, 2, 3$. These digits are then classified using a k-Nearest Neighbour algorithm, which is trained on the provided training digit images.
    - The spatial occupancy feature vector of the associated arrow region is calculated $v(R_{i_4}) \in \mathbb{R}^{2\times2}$ with 2 x-bins and 2 y-bins. This arrow is then classified using a k-Nearest Neighbour algorithm, which is trained on the provided training arrow images.

This detection algorithm is demonstrated in Figure 6.



(a) *(b)*

(c) *(d)*

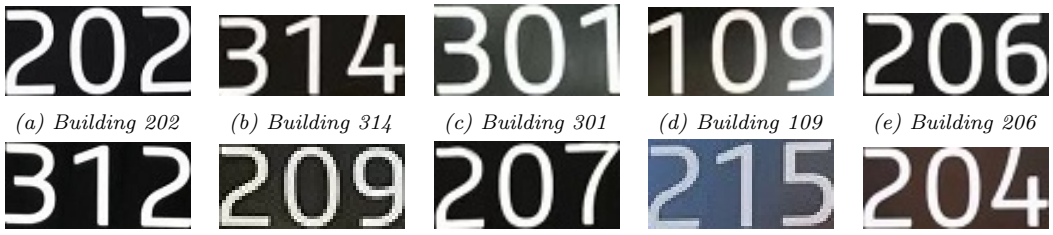
Figure 6: Demonstration of the detection algorithm for task 2, applied to `task2/val03.jpg`*: a - Chains of regions detected; b - Largest set of aligned chains detected; c - Missing digits detected; d - Associated arrows detected.*

3 Validation Performance

The performance of the detection and classification algorithms for task 1 and task 2 on the validation images are shown in subsection 3.1 and subsection 3.2 respectively. It is seen that the detection algorithms are quite effective, handling well various lightning conditions, camera perspectives, and the presence of noisy natural scenes. Furthermore, the k-NN classification algorithms seem to be work appropriately. In fact, no error is seen at all on the validation sets remarkably.

3.1 Task 1

The performance of the detection and classification algorithms on the validation image set, for task 1, is presented in Figure 7.



(a) Building 202 *(b) Building 314* *(c) Building 301* *(d) Building 109* *(e) Building 206*

(f) Building 312 *(g) Building 209* *(h) Building 207* *(i) Building 215* *(j) Building 204*

Figure 7: Detected regions of interest, and predicted digits for each of the validation images for task 1.

3.2 Task 2

The performance of the detection and classification algorithms on the validation image set, for task 2, is presented in Figure 8.

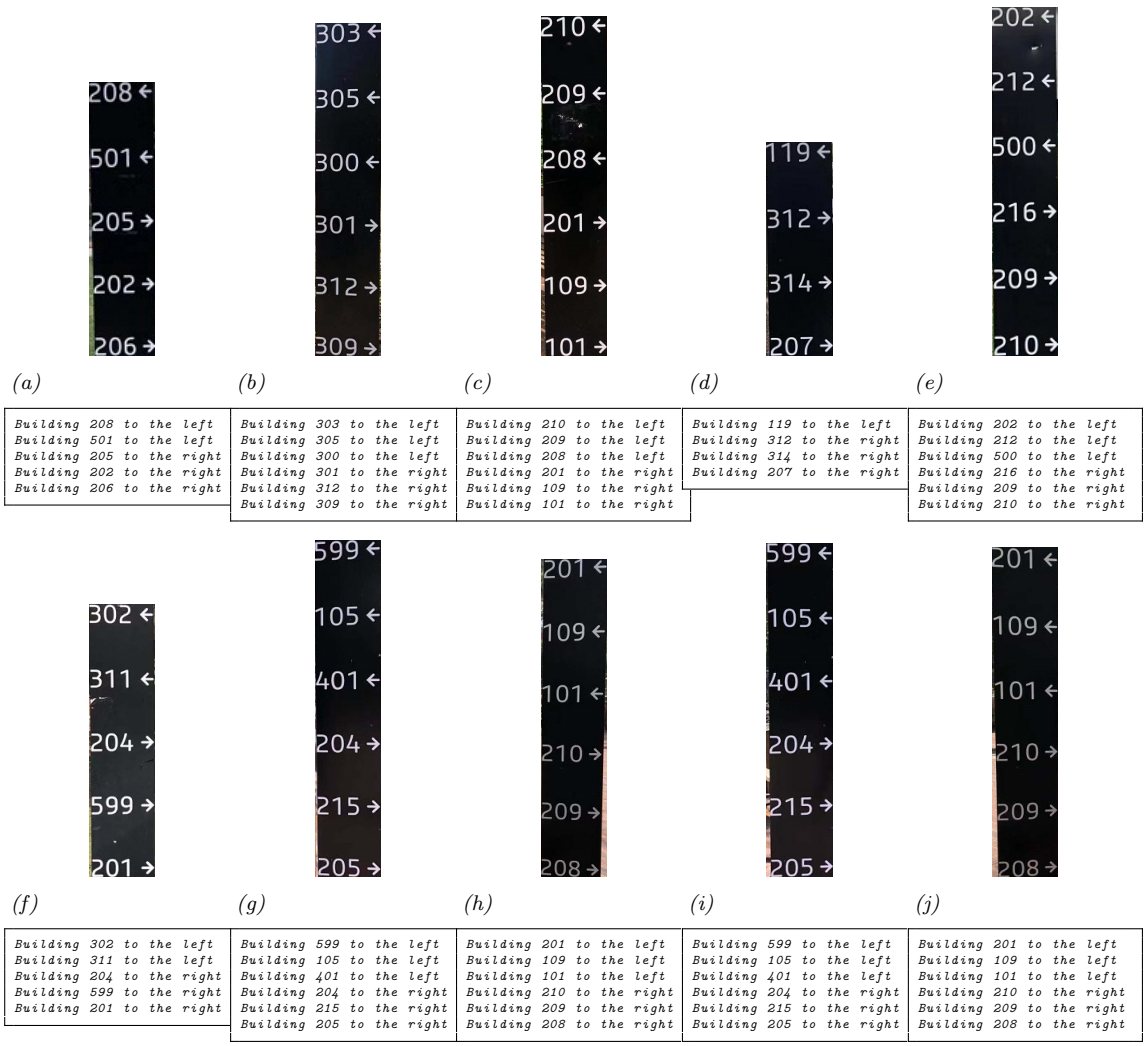


Figure 8: Detected regions of interest, and predicted digits for each of the validation images for task 2.

References

[Matas et al., 2004] Matas, J., Chum, O., Urban, M., and Pajdla, T. (2004). Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761–767. British Machine Vision Computing 2002.

A Source Code

A.1 box.py

```
#!/usr/bin/env python3

import numpy as np
import cv2


class Box:
    """
    Rectangular box, suitable for use with OpenCV.

    Attributes
    ----------
    x : int
        x-coordinate of top-left corner
    y : int
        y-coordinate of top-left corner
    width : int
        x-length of box
    height : int
        y-length of box

    Methods
    -------
    tl : (int, int)
        Top left corner of the box.

    br : (int, int)
        Bottom right corner of the box.

    tr : (int, int)
        Top right corner of the box.

    bl : (int, int)
        Bottom left corner of the box.

    center : (int, int)
        Geometric center point of the box.

    area : int
        Rectangular area of the box.

    aspect : float
        Aspect ratio of the box, in height/width format.

    indexes : (slice int, slice int)
```

```
46          2-D array index slices that this box corresponds to.
47          Suitable for accessing 2-D arrays as `array[box.indexes]`.
48
49      contains(point) : bool
50          Returns true if `point` is within the box.
51
52      overlap(box) : float
53          The fraction of the area of `box` that overlaps with this box.
54
55      is_superset_of(box) : bool
56          Returns true if `box` is entirely inside this box.
57
58      Notes
59      -----
60      As per OpenCV, a point `(x, y)` corresponds to an array index `array[y, x]`.
61
62      """
63
64      def __init__(self, x, y, width, height):
65          self._x = x
66          self._y = y
67          self._width = width
68          self._height = height
69
70      @property
71      def x(self):
72          return self._x
73
74      @property
75      def y(self):
76          return self._y
77
78      @property
79      def width(self):
80          return self._width
81
82      @property
83      def height(self):
84          return self._height
85
86      @property
87      def tl(self):
88          return (self.x, self.y)
89
90      @property
91      def br(self):
92          return (self.x + self.width, self.y + self.height)
93
94      @property
95      def tr(self):
96          return (self.x + self.width, self.y)
97
98      @property
99      def bl(self):
100         return (self.x, self.y + self.height)
101
102     @property
103     def center(self):
```

```
104         return (self.x + int(self.width / 2), self.y + int(self.height / 2))
105
106     @property
107     def area(self):
108         return (self.width * self.height)
109
110     @property
111     def aspect(self):
112         return (self.height / self.width)
113
114     @property
115     def indexes(self):
116         slice_x = slice(self.x, self.x + self.width)
117         slice_y = slice(self.y, self.y + self.height)
118         return (slice_y, slice_x)
119
120     def contains(self, point):
121         x, y = point[0], point[1]
122         return (self.x <= x <= self.x + self.width
123                 and self.y <= y <= self.y + self.height)
124
125     def overlap(self, box):
126         a_x = max([self.x, box.x])
127         b_x = min([self.x + self.width, box.x + box.width])
128         w = b_x - a_x if b_x > a_x else 0
129
130         a_y = max([self.y, box.y])
131         b_y = min([self.y + self.height, box.y + box.height])
132         h = b_y - a_y if b_y > a_y else 0
133         return ((w * h) / box.area)
134
135     def is_superset_of(self, box):
136         return (self.x <= box.x <= self.x + self.width - box.width
137                 and self.y <= box.y <= self.y + self.height - box.height)
138
139     def __str__(self):
140         properties = \
141             f"Box:\n"\
142             + f"tl = ({self.x}, {self.y})\n"\
143             + f"width = {self.width}\n"\
144             + f"height = {self.height}\n"\
145             + f"area = {self.area}\n"\
146             + f"aspect = {self.aspect}"
147         return properties
148
149
150 def covering_box(boxes):
151     """
152     Construct the smallest box which covers a collection of boxes.
153
154     Parameters
155     ----------
156     boxes : iterable collection of Box
157
158     Returns
159     -------
160     cover : Box
161
```

```
162     """
163     x_min = np.amin([b.x for b in boxes])
164     x_max = np.amax([b.x + b.width for b in boxes])
165     y_min = np.amin([b.y for b in boxes])
166     y_max = np.amax([b.y + b.height for b in boxes])
167     cover = Box(x_min, y_min, x_max - x_min, y_max - y_min)
168     return cover
169
170
171 def bounding_box(points):
172     """
173     Construct the minimal bounding box for a given set of 2-D points.
174
175     Parameters
176     ----------
177     points : iterable collection of (int, int)
178
179     Returns
180     -------
181     bounding : Box
182
183     """
184     x, y, w, h = cv2.boundingRect(np.array([p for p in points]))
185     bounding = Box(x, y, w, h)
186     return bounding
187
188
189 def merge_overlapping(boxes, max_overlap=0.05):
190     """
191     Merge all sufficiently overlapping boxes in a collection of boxes.
192
193     Parameters
194     ----------
195     boxes : iterable collection of Box
196     max_overlap : float, default=0.05
197         Merge any pair of boxes for which one of them overlaps the other more
198         than this value.
199
200     Returns
201     -------
202     boxes_merged : list of Box
203
204     """
205     def overlaps(bi, bj):
206         return (bi.overlap(bj) >= max_overlap
207                 or bj.overlap(bi) >= max_overlap)
208
209     def merge_into(boxes, box):
210         overlapping = [b for b in boxes if overlaps(box, b)]
211         if (len(overlapping) == 0):
212             return (boxes + [box])
213         else:
214             preserved = [b for b in boxes if not overlaps(box, b)]
215             merged = covering_box(overlapping + [box])
216             return (merge_into(preserved, merged))
217
218     boxes_merged = []
219     for b in boxes:
```

```
220             boxes_merged = merge_into(boxes_merged, b)
221         return boxes_merged
222
223
224 def otsu_separation(img_gray, box):
225     """
226     Calculate the Otsu separation of a single-channel image restricted to a box.
227
228     Parameters
229     ----------
230     img_gray : 2-D array of int
231         Single channel image.
232     box : Box
233         The 2-D restriction of `img_gray` for which the Otsu separation is
234         calculated.
235
236     Returns
237     -------
238     otsu_sep : float
239         The separation between the black and white class means, after
240         determining class by Otsu thresholding.
241
242     """
243     img_box = (img_gray[box.indexes]).astype(np.uint8)
244     t, img_bin = cv2.threshold(img_box, 128, 255, cv2.THRESH_OTSU)
245
246     idxs_w = img_box > t
247     idxs_b = img_box <= t
248
249     mean_w = np.average(img_box[idxs_w])
250     mean_b = np.average(img_box[idxs_b])
251     otsu_sep = mean_w - mean_b
252     return otsu_sep
253
254
255 def otsu_separation_color(img, box):
256     """
257     Calculate the Otsu separation of a colour image restricted to a box.
258
259     Parameters
260     ----------
261     img : 3-D array of int
262         Color image with 3 colour channels.
263     box : Box
264         The 2-D restriction of `img` for which the Otsu separation is
265         calculated.
266
267     Returns
268     -------
269     min_otsu_sep : float
270         The minimum of the Otsu separations calculated for each colour channel
271         of `img` and its grayscale transformation.
272
273     """
274     img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
275     min_otsu_sep = np.amin(
276         [otsu_separation(img_gray, box),
277          otsu_separation(img[:, :, 0], box),
```

```
278             otsu_separation(img[:, :, 1], box),
279             otsu_separation(img[:, :, 2], box)])
280     return min_otsu_sep
```

A.2 region.py

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4  import cv2
5  import math
6  import random
7
8  from box import *
9
10
11 class Region:
12     """
13     Connected region of points, suitable for use with OpenCV MSER.
14
15     Attributes
16     ----------
17     points : set of (int, int)
18         Set of points that this region contains.
19         These points are assumed, but not checked, upon construction to be
20         connected.
21
22     box : Box
23         Minimal bounding box of this region.
24
25     boundary : set of (int, int)
26         Set of points of this region which are adjacent to at least point not in
27         this region.
28         Calculated using OpenCV's `findContours()` method.
29         Is calculated as needed, and then cached.
30
31     cached_boundary : bool
32         Flag true if the boundary points have been calculated.
33
34     contours : list of list of (int, int)
35         The set of contours calculated using OpenCV's `findContours()` method.
36         Is calculated as needed, and then cached.
37
38     hierarchy : 2-D array
39         The heirarchy of the set of contours calculated using OpenCV's
40         `findContours()` method.
41         Is calculated as needed, and then cached.
42
43     cached_contours : bool
44         Flag true if the contours have been calculated.
45
46     Methods
47     -------
48     area : int
49         The cardinality of `points`; that is, the number of points this region
50         contains.
```

```
51
52      fill : float
53          The fraction of the area of the bounding box of this region that this
54          region also contains.
55
56      holes : int
57          The number of interior holes this regions contains.
58          Inferred from the number of contours this region has.
59
60      moments : dict of (string, float)
61          The image moments of this region, assuming each point has a mass of 1.
62          Calculated using OpenCV's `moments()` method.
63
64      centroid : (float, float)
65          The mass centre of this region, calculated from its moments.
66
67      hu_moments : array of float
68          The Hu moments of this region, calculated using OpenCV's `HuMoments()`
69          method.
70
71      image : 2-D array of int
72          Represents this region as binary image, with dimensions of its bounding
73          box, with a point in the image being white if it is in `points`.
74
75      spatial_occupancy(bins_x, bins_y) : 2-D array of float
76          Constructs a set of bins, symmetric about the geometric centre of `box`,
77          with `bins_x` x-bins and `bins_y` y-bins, then calculates the fill of
78          each bin by the region.
79
80      distance(point) : float
81          Calculates the minimum distance of `point` to any of the points in
82          this region.
83
84      set_distance_min(points) : float
85          Calculates the minimum of all distances between `points` and points in
86          this region.
87
88      set_distance_max(points) : float
89          Calculates the maximum of all distances between `points` and points in
90          this region.
91
92      overlap(region) : float
93          Calculates the fractional cardinality of the intersection of `region`
94          with this region, over the cardinality of `region`.
95
96      contains(region) : bool
97          Returns true if all points in `region` are also in this region.
98
99      """
100
101     def __init__(self, points):
102         self._points = set([(p[0], p[1]) for p in points])
103         self._box = bounding_box(self.points)
104
105         self._cached_boundary = False
106         self._boundary = None
107
108         self._cached_contours = False
```

```
109         self._contours = None
110         self._hierarchy = None
111 
112     @property
113     def points(self):
114         return self._points
115 
116     @property
117     def box(self):
118         return self._box
119 
120     @property
121     def area(self):
122         return len(self.points)
123 
124     @property
125     def fill(self):
126         return (self.area / self.box.area)
127 
128     @property
129     def boundary(self):
130         if not self._cached_boundary:
131             cs = [np.reshape(c, (-1, 2)) for c in (self.contours[0])]
132             self._boundary = set(
133                 [(self.box.x + p[0], self.box.y + p[1])
134                  for p in np.concatenate(cs)])
135             self._cached_boundary = True
136         return self._boundary
137 
138     @property
139     def holes(self):
140         return (len(self.contours[0]) - 1)
141 
142     @property
143     def moments(self):
144         return cv2.moments(self.image().astype(np.float32), binaryImage=True)
145 
146     @property
147     def centroid(self):
148         m = self.moments
149         return (m["m10"] / m["m00"], m["m01"] / m["m00"])
150 
151     @property
152     def hu_moments(self):
153         return cv2.HuMoments(self.moments)[:, 0]
154 
155     @property
156     def contours(self):
157         if not self._cached_contours:
158             _, self._contours, self._hierarchy = cv2.findContours(
159                 self.image().astype(np.uint8),
160                 cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
161             self._cached_contours = True
162         return (self._contours, self._hierarchy)
163 
164     def image(self):
165         img = np.zeros((self.box.height, self.box.width), dtype=np.uint8)
166         for point in self.points:
```

```
167             j, i = (point[0] - self.box.x, point[1] - self.box.y)
168             img[i, j] = 255
169         return img
170
171     def spatial_occupancy(self, bins_x, bins_y):
172         s_x = math.ceil(self.box.width / bins_x)
173         s_y = math.ceil(self.box.height / bins_y)
174
175         c_x = math.floor((self.box.width - ((bins_x - 2) * s_x)) / 2)
176         c_y = math.floor((self.box.height - ((bins_y - 2) * s_y)) / 2)
177
178         xs = np.array(
179             [0]
180             + [(i * s_x) + c_x for i in range(0, bins_x - 1)]
181             + [self.box.width])
182         ys = np.array(
183             [0]
184             + [(i * s_y) + c_y for i in range(0, bins_y - 1)]
185             + [self.box.height])
186
187         img = self.image()
188         bins = np.zeros((bins_y, bins_x), dtype=np.float32)
189         for i in range(bins_y):
190             sl_y = slice(ys[i], ys[i+1])
191             for j in range(bins_x):
192                 sl_x = slice(xs[j], xs[j+1])
193                 n = (ys[i+1] - ys[i]) * (xs[j+1] - xs[j])
194                 bins[i, j] = np.count_nonzero(img[sl_y, sl_x]) / n
195         return bins
196
197     def distance(self, point):
198         if point in self.points:
199             min_distance = 0.0
200         else:
201             norm = lambda p: np.sqrt(np.abs((p[0] ** 2) + (p[1] ** 2)))
202             diff = lambda p1, p2: (p1[0] - p2[0], p1[1] - p2[1])
203
204             min_distance = np.amin(
205                 [norm(diff(bp, point)) for bp in self.boundary])
206         return min_distance
207
208     def set_distance_min(self, points):
209         return np.amin([self.distance(p) for p in points])
210
211     def set_distance_max(self, points):
212         return np.amax([self.distance(p) for p in points])
213
214     def overlap(self, region):
215         return len(self.points.intersection(region.points)) / len(region.points)
216
217     def contains(self, region):
218         return np.all([(p in self.points) for p in region.points])
219
220     def show(self):
221         cv2.imshow("region", self.image())
222         cv2.waitKey(0)
223         cv2.destroyWindow("region")
224         return
```

```
225
226     def __str__(self):
227         properties = \
228             f"Region:\n"\
229             + f"{str(self.box)}\n"\
230             + f"area = {self.area}\n"\
231             + f"fill = {self.fill}\n"\
232             + f"holes = {self.holes}"
233         return properties
234
235
236 def remove_overlapping(regions, max_overlap=0.8):
237     """
238     Filters regions by removing sufficiently overlapping smaller regions.
239
240     Parameters
241     ----------
242     regions : iterable collection of Region
243     max_overlap : float, default=0.8
244         Remove any region which overlaps with a larger region, by area, by more
245         than this value.
246
247     Returns
248     -------
249     regions_filtered : iterable collection of Region
250
251     """
252     regions_ordered = sorted(regions, key=lambda r: r.area, reverse=True)
253     regions_filtered = []
254     for r in regions_ordered:
255         if np.all([not (rf.box.is_superset_of(r.box)
256                         and rf.overlap(r) >= max_overlap)
257                    for rf in regions_filtered]):
258             regions_filtered.append(r)
259     return regions_filtered
260
261
262 def remove_occluded_holes(regions, max_boundary_distance=10):
263     """
264     Filters interior hole regions, which fill up another regions hole.
265
266     Parameters
267     ----------
268     regions : iterable collection of Region
269     max_boundary_distance : int, default=10
270         Remove any region with boundary points which are never more than this
271         distance away from another region which contains this one.
272
273     Returns
274     -------
275     regions_filtered : iterable collection of Region
276
277     """
278     regions_ordered = sorted(regions, key=lambda r: r.box.x)
279     regions_filtered = []
280     for r in regions_ordered:
281         occludes = lambda rf: np.all(
282             [rf.distance(bp) <= max_boundary_distance for bp in r.boundary])
```

```
283         if np.all([not (rf.box.is_superset_of(r.box) and occludes(rf))
284                    for rf in regions_filtered]):
285             regions_filtered.append(r)
286     return regions_filtered
287
288
289 def draw_regions(regions, size=None):
290     """
291     Creates an image from a set of regions.
292
293     Parameters
294     ----------
295     regions : iterable collection of Region
296     size : (int, int), optional
297         Height and width of the image canvas, on which to draw the regions.
298
299     Returns
300     -------
301     img_regions : 3-D array of int
302         Colour image (constructed in HSV space, but returned in BGR) with a
303         black background, distinct colours for each region, and with the
304         boundaries of regions coloured white.
305
306     """
307     if size:
308         canvas = Box(0, 0, size[1], size[0])
309     else:
310         canvas = covering_box([r.box for r in regions])
311
312     img_regions = np.zeros(
313         (canvas.height, canvas.width, 3), dtype=np.uint8)
314     for r in regions:
315         color = (random.randint(0, 179), 255, 255)
316         for p in r.points:
317             x, y = p
318             img_regions[y - canvas.y, x - canvas.x] = color
319         for bp in r.boundary:
320             x, y = bp
321             img_regions[y - canvas.y, x - canvas.x] = (0, 0, 255)
322     img_regions = cv2.cvtColor(img_regions, cv2.COLOR_HSV2BGR)
323     return img_regions
324
325
326 def cc_regions(img_bin):
327     """
328     Creates a set of regions from the connected components of a binary image.
329
330     Parameters
331     ----------
332     img_bin : 2-D array of int
333         Binary image.
334
335     Returns
336     -------
337     regions : list of Region
338         The set of regions formed from connected components of a binary image.
339
340     """
```

```
341     n, labels = cv2.connectedComponents(img_bin, connectivity=8)
342     regions = [Region(np.argwhere(np.transpose(labels) == l))
343                for l in range(1, n)]
344     return regions
```

A.3 chain.py

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4  import cv2
5
6  from box import *
7  from region import *
8
9
10 def linked(region_1, region_2):
11     """
12     Determine if two regions are sufficiently adjacent and similar.
13
14     Parameters
15     ----------
16     region_1 : Region
17     region_2 : Region
18
19     Returns
20     -------
21     bool
22         True if the bounding boxes of `region_1` and `region_2`
23         - have similar `y` placement; and
24         - have similar `height`; and
25         - have adjacent `x` placement; and
26         - do not overlap too much.
27
28     """
29
30     # these are ratios with regard to box heights
31     max_ratio_diff_y = 0.5
32     max_ratio_diff_height = 0.2
33     max_ratio_diff_x = 1.0
34     max_overlap = 0.25
35
36     diff_x = np.abs(region_2.box.x - region_1.box.x)
37     diff_y = np.abs(region_2.box.y - region_1.box.y)
38     diff_height = np.abs(region_2.box.height - region_1.box.height)
39
40     similar_height = (
41         diff_height <= max_ratio_diff_height*region_1.box.height
42         and diff_height <= max_ratio_diff_height*region_2.box.height)
43
44     similar_y = (
45         diff_y <= max_ratio_diff_y*region_1.box.height
46         and diff_y <= max_ratio_diff_y*region_2.box.height)
47
48     adjacent_x = (
49         diff_x <= max_ratio_diff_x*region_1.box.height
```

```
50          and diff_x <= max_ratio_diff_x*region_2.box.height)
51
52      non_occluding = (
53          (not region_1.box.is_superset_of(region_2.box))
54          and region_1.box.overlap(region_2.box) <= max_overlap
55          and region_2.box.overlap(region_1.box) <= max_overlap)
56
57      return (similar_height and similar_y and adjacent_x and non_occluding)
58
59
60  def find_chains(regions, best_edge=True):
61      """
62      Find all non-overlapping chains of regions, in a set of regions.
63
64      Finds all links between regions, then filters the edges by distance between
65      to regions to ensure each region has at most one adjacent region to their
66      right and at most one adjacent region to their left - ensuring all paths are
67      non-overlapping.
68      All paths are then extracted and are said to be chains of regions.
69
70      Parameters
71      ----------
72      regions : list of Region
73      best_edge : bool, default=True
74          Disabling this results in the edges not being filtered, and so
75          overlapping paths may be extracted.
76
77      Returns
78      -------
79      list of list of Region
80          List of chains (represented as a list of regions) of adjacent regions.
81
82      """
83      regions_ordered = sorted(regions, key=lambda r: r.box.x)
84      n = len(regions_ordered)
85
86      edges = dict()
87      roots = set(range(0, n))
88      for i in range(n):
89          ri = regions_ordered[i]
90
91          links = {j for j in range(i+1, n) if linked(ri, regions_ordered[j])}
92          if best_edge:
93              links_sorted = sorted(
94                  links,
95                  key=(lambda j: ri.distance(regions_ordered[j].box.center)))
96              links = set(links_sorted[:1])
97
98          edges[i] = links
99          roots -= edges[i]
100         if not edges[i]:
101             roots -= {i}
102
103     def paths(i):
104         if edges[i]:
105             suffixes = [p for j in edges[i] for p in paths(j)]
106             return [[i] + s for s in suffixes]
107         else:
```

```
108             return [[i]]
109
110     chains = [p for i in roots for p in paths(i)]
111     return [list(map(lambda i: regions_ordered[i], c)) for c in chains]
112
113
114 def cluster_largest_otsu_separations(img, chains, max_diff=50):
115     """
116     Filters a set of chains, to leave the most monochromatic chains.
117
118     For each chain, the minimum Otsu separation of the image across all channels
119     (RGB and grayscale), restricted to the bounding box of the chain, is
120     calculated.
121     The chains are sorted by their separation, and the cluster of most separated
122     chains is taken.
123
124     Parameters
125     ----------
126     img : 3-D array of int
127         Colour image.
128     chains : list of list of Region
129     max_diff : int, default=50
130         The cluster of most separated chains is parameterised by this value;
131         once ordered by decreasing separation, chains are taken until the change
132         in separation from one chain to the next is more than `max_diff`.
133
134     Returns
135     -------
136     list of list of Region
137         The list of chains, ordered by decreasing Otsu separation, which are
138         more separated, by `max_diff`, than the remaining list of chains.
139
140     """
141     if not chains:
142         return chains
143
144     chains_ordered = sorted(
145         chains,
146         key=(lambda c: \
147              otsu_separation_color(img, covering_box([r.box for r in c]))),
148         reverse=True)
149
150     otsu_seps = np.array(
151         [otsu_separation_color(img, covering_box([r.box for r in c]))
152          for c in chains_ordered])
153
154     n = len(otsu_seps)
155     idx = n-1
156     for i in range(n-1):
157         diff = np.abs(otsu_seps[i+1] - otsu_seps[i])
158         if diff >= max_diff:
159             idx = i
160             break
161
162     return (chains_ordered[:idx+1])
163
164
165 def aligned(chain_1, chain_2):
```

```
166     """
167     Determine if two chains (of at most 3 regions) are aligned vertically.
168
169     The two chains are considered to be aligned if each of their two right-most
170     regions (expected to be digits) are:
171     - aligned vertically (or near vertically); and
172     - of correspondingly similar heights.
173
174     Parameters
175     ----------
176     chain_1 : list of Region
177     chain_2 : list of Region
178
179     Returns
180     -------
181     bool
182         Flag true if the two chains are aligned.
183
184     """
185     n1 = len(chain_1)
186     n2 = len(chain_2)
187
188     if (not (2 <= n1 <= 3)) and (not (2 <= n2 <= 3)):
189         return False
190
191     norm = lambda p: np.sqrt(np.abs((p[0] ** 2) + (p[1] ** 2)))
192     diff = lambda p1, p2: (p1[0] - p2[0], p1[1] - p2[1])
193
194     box_1 = covering_box([r.box for r in chain_1])
195     box_2 = covering_box([r.box for r in chain_2])
196     w = min([box_1.width, box_2.width])
197
198     n = min([n1, n2])
199     digits_1 = {i: chain_1[n1-i-1] for i in range(n)}
200     digits_2 = {i: chain_2[n2-i-1] for i in range(n)}
201
202     aligned_vert = np.all(
203         [np.abs(diff(digits_1[i].box.center, digits_2[i].box.center)[0])
204          <= w
205          for i in range(n)])
206
207     similar_heights = np.all(
208         [np.abs(digits_1[i].box.height - digits_2[i].box.height)
209          <= 0.2 * min([digits_1[i].box.height, digits_2[i].box.height])
210          for i in range(n)])
211
212     return (aligned_vert and similar_heights)
213
214
215 def find_aligned_chains(chains):
216     """
217     Find the largest subset of aligned chains, from a set of chains.
218
219     Each pair of chains is compared for alignment, forming a graph of chains.
220     This graph is then searched (depth-first) to form a partition of `chains`
221     which groups chains by alignment, and connectedness thereof.
222     The largest subset of aligned chains is assumed to be the subset of
223     interest, and is returned.
```

```
224
225     Parameters
226     ----------
227     chains : list of list of Region
228
229     Returns
230     -------
231     list of list of Region
232         The largest subset of chains which are aligned.
233
234     """
235     chains_ordered = sorted(
236         chains, key=lambda c: covering_box([r.box for r in c]).y)
237     n = len(chains_ordered)
238
239     edges = dict()
240     for i, ci in enumerate(chains_ordered[0:n]):
241         edges[i] = {j for j, cj in enumerate(chains_ordered)
242                     if (j != i and aligned(ci, cj))}
243
244     def dfs(explored, i):
245         idxs = {i}
246         if i not in explored:
247             explored |= {i}
248             for j in edges[i]:
249                 idxs |= dfs(explored, j)
250         return idxs
251
252     k = 0
253     eq_classes = dict()
254     explored = set()
255     for i in range(n):
256         if i not in explored:
257             eq_classes[k] = dfs(explored, i)
258             k += 1
259
260     aligned_chains = [[chains_ordered[j]
261                        for j in eq_classes[k]]
262                       for k in eq_classes
263                       if len(eq_classes[k]) > 1]
264
265     return max(aligned_chains, key=lambda ac: len(ac))
266
267
268 def find_missing_digits(aligned_chains, img_gray):
269     """
270     Find any missing left digits in a set of aligned chains.
271
272     For each chain, of only two regions, in a set of aligned chains, the
273     bounding box of the missing left digit is estimated.
274     The grayscale image is then restricted to this box, thresholded to a binary
275     image, and the connected components are calculated.
276     The largest connected component is assumed to be the missing digit, and
277     prepended to the chain.
278
279     Parameters
280     ----------
281     aligned_chains : list of list of Region
```

```
282         List of chains (of two or three regions) with possibly missing left
283         digits.
284     img_gray : 2-D array of int
285         Grayscale image, to be used in the construction an approximate region
286         for the missing digits.
287
288     Returns
289     -------
290     aligned_chains_found : list of list of Region
291         List of chains (of exactly three regions) with all missing left digits
292         localised and found.
293
294     """
295     H, W = img_gray.shape[:2]
296
297     aligned_chains_found = []
298     for chain in aligned_chains:
299         n = len(chain)
300
301         if n == 3:
302             aligned_chains_found.append(chain)
303             continue
304
305         digit_2 = chain[n-2]
306         digit_3 = chain[n-1]
307
308         box = covering_box([digit_2.box, digit_3.box])
309         diff_x = digit_3.box.tl[0] - digit_2.box.tl[0]
310         diff_y = digit_3.box.tl[1] - digit_2.box.tl[1]
311
312         x = max([0, box.tl[0] - int(0.9*diff_x)])
313         y = min([H - 1, max([0, box.tl[1] - diff_y])])
314         w = box.tl[0] - x - 1
315         h = min([H - y - 1, max([digit_2.box.height, digit_3.box.height])])
316         left_box = Box(x, y, w, h)
317
318         img_box = (img_gray[left_box.indexes]).astype(np.uint8)
319         t, img_bin = cv2.threshold(img_box, 128, 255, cv2.THRESH_OTSU)
320         regions = cc_regions(img_bin)
321
322         region = max(regions, key=lambda r: r.area)
323         digit_1 = Region({(x + p[0], y + p[1]) for p in region.points})
324
325         aligned_chains_found.append([digit_1, digit_2, digit_3])
326     return aligned_chains_found
327
328
329 def find_arrows(aligned_chains, regions):
330     """
331     Find the directional arrows associated with each chain for a set of chains.
332
333     For each chain, the bounding box of two rightmost digits is used to
334     construct an estimate for a box which will contain the directional arrow
335     region, and likely no other region.
336     The region corresponding to the arrow is assumed to be contained maximally
337     within this box.
338
339     Parameters
```

```
340         ----------
341         aligned_chains : list of list of Region
342             List of aligned chains, assumed to be a sequence of 2 or 3 digits, for
343             which the associated arrows are to be found.
344         regions : list of Region
345             List of regions which is assumed to contain the regions corresponding to
346             the directional arrows.
347
348         Returns
349         -------
350         aligned_chains_arrows : list of (list of Region, Region)
351             List of chains (of digits) and their associated directional arrow region.
352
353         """
354         aligned_chains_arrows = []
355         for chain in aligned_chains:
356             n = len(chain)
357
358             digit_2 = chain[n-2]
359             digit_3 = chain[n-1]
360
361             box = covering_box([digit_2.box, digit_3.box])
362             right_box = Box(box.tr[0] + 1, box.tr[1], box.width, box.height)
363
364             arrow = max(regions, key=lambda r: right_box.overlap(r.box))
365             aligned_chains_arrows.append((chain, arrow))
366         return aligned_chains_arrows
```

A.4 knn.py

```
1  #!/usr/bin/env python3
2
3  import os
4  import numpy as np
5  import cv2
6
7  from region import *
8
9
10 class KNN:
11     """
12     k-Nearest Neighbour class, built around OpenCV's KNearest object.
13
14     Attributes
15     ----------
16     labels : dict of (int, X), where X is type of label of samples
17         A map between the int labels used internally, and the labels of the
18         samples as provided to the `train()` method.
19     knn : cv2.ml.KNearest
20         The k-Nearest Neighbour object, trained on the sample data, which is
21         used to predict class labels for unlabelled sample data.
22
23     Methods
24     -------
25     train(samples_labelled) :
26         Train `knn` on the labelled sample data, and build the map `labels`,
```

```
27          between internal labels for OpenCV's `train()` and `predict()` methods,
28          and the labels of `samples_labelled`.
29
30      predict(samples, k=3) :
31          Predict the class labels of `samples` using `k` neighbours, and the
32          trained `knn` object.
33          The output labels are converted from the internal class labels to be of
34          the same type as was provided during training.
35
36      """
37
38      def __init__(self, samples_labelled):
39          self.train(samples_labelled)
40          return
41
42      @property
43      def labels(self):
44          return self._labels
45
46      @property
47      def knn(self):
48          return self._knn
49
50      def train(self, samples_labelled):
51          self._labels = {k : l for k, l in enumerate(samples_labelled.keys())}
52
53          samples = np.concatenate(
54              [samples_labelled[self.labels[k]].astype(np.float32)
55               for k in self.labels])
56
57          responses = np.concatenate(
58              [np.full((samples_labelled[self.labels[k]].shape[0]), k)
59               for k in self.labels])
60
61          self._knn = cv2.ml.KNearest_create()
62          self._knn.setIsClassifier(True)
63          self._knn.setAlgorithmType(cv2.ml.KNearest_BRUTE_FORCE)
64          self._knn.train(samples, cv2.ml.ROW_SAMPLE, responses)
65          return
66
67      def predict(self, samples, k=3):
68          _, responses, _, dist = self.knn.findNearest(samples, k)
69
70          labels_predicted = np.array(
71              [self.labels[np.int32(r[0])]
72               for i, r in enumerate(responses)])
73
74          return labels_predicted
75
76
77  def build_knn_digits(dir_digits, bins_x, bins_y):
78      """
79      Construct a KNN object from a directory containing digit training images.
80
81      Parameters
82      ----------
83      dir_digits : string
84          Path for the directory containing the digit training images.
```

```
85      bins_x : int
86          Number of x-component bins for `spatial_occupancy()`.
87      bins_y : int
88          Number of y-component bins for `spatial_occupancy()`.
89
90      Returns
91      -------
92      KNN
93          k-Nearest Neighbour algorithm trained on the digit images provided.
94
95      """
96      digits = {
97          0: "Zero",
98          1: "One",
99          2: "Two",
100         3: "Three",
101         4: "Four",
102         5: "Five",
103         6: "Six",
104         7: "Seven",
105         8: "Eight",
106         9: "Nine"}
107
108     imgs = dict()
109     for d in iter(digits):
110         for k in range(5):
111             digit_file = os.path.join(dir_digits, f"{digits[d]}{k+1}.jpg")
112             imgs[(d, k)] = cv2.imread(digit_file, cv2.IMREAD_COLOR)
113
114     samples = dict()
115     for d in iter(digits):
116         samples[d] = np.zeros((5, bins_x * bins_y), dtype=np.float32)
117         for k in range(5):
118             img_gray = cv2.cvtColor(imgs[(d, k)], cv2.COLOR_BGR2GRAY)
119             _, img_bin = cv2.threshold(img_gray, 128, 255, cv2.THRESH_OTSU)
120
121             regions = cc_regions(img_bin)
122             h, w = img_bin.shape[:2]
123             region = min(
124                 regions, key=lambda r: r.distance((int(w/2), int(h/2))))
125
126             samples[d][k] = np.ravel(region.spatial_occupancy(bins_x, bins_y))
127
128     return KNN(samples)
129
130
131 def build_knn_arrows(dir_arrows, bins_x, bins_y):
132     """
133     Construct a KNN object from a directory containing arrow training images.
134
135     Parameters
136     ----------
137     dir_arrows : string
138         Path for the directory containing the arrow training images.
139     bins_x : int
140         Number of x-component bins for `spatial_occupancy()`.
141     bins_y : int
142         Number of y-component bins for `spatial_occupancy()`.
```

```
143
144     Returns
145     -------
146     KNN
147         k-Nearest Neighbour algorithm trained on the arrow images provided.
148
149     """
150     arrows = {
151         "L": "LeftArrow",
152         "R": "RightArrow"}
153
154     imgs = dict()
155     for a in iter(arrows):
156         for k in range(5):
157             arrow_file = os.path.join(dir_arrows, f"{arrows[a]}{k+1}.jpg")
158             imgs[(a, k)] = cv2.imread(arrow_file, cv2.IMREAD_COLOR)
159
160     samples = dict()
161     for a in iter(arrows):
162         samples[a] = np.zeros((5, bins_x * bins_y), dtype=np.float32)
163         for k in range(5):
164             img_gray = cv2.cvtColor(imgs[(a, k)], cv2.COLOR_BGR2GRAY)
165             _, img_bin = cv2.threshold(img_gray, 128, 255, cv2.THRESH_OTSU)
166
167             regions = cc_regions(img_bin)
168             h, w = img_bin.shape[:2]
169             region = min(
170                 regions, key=lambda r: r.distance((int(w/2), int(h/2))))
171
172             samples[a][k] = np.ravel(region.spatial_occupancy(bins_x, bins_y))
173
174     return KNN(samples)
```

A.5 parser.py

```
1  #!/usr/bin/env python3
2
3  import os
4  import argparse
5  import re
6  import cv2
7
8
9  def parse_input():
10     """
11     Parse command line arguments.
12
13     Returns
14     -------
15     args : dict of (string, values)
16         Dictionary of command line arguments.
17         - args["input"] is the directory of input images to detect and classify;
18         - args["output"] is the directory for output images and txt files to be
19           written to;
20         - args["digits"] is the directory of digit and directional arrow
21           training images;
```

```
22          - args["work"] is the directory that work (pipeline) images will be
23            written to;
24          - args["work_save"] is a flag indicating if work images are to be
25            constructed and saved, or not.
26      img_files : list of string
27          The set of input images on which the detection and classification
28          algorithms are to be run.
29
30      """
31      parser = argparse.ArgumentParser()
32      parser.add_argument("-i", "--input", required=True,
33                          help="directory path with input images")
34      parser.add_argument("-o", "--output", required=True,
35                          help="directory path for output images and data")
36      parser.add_argument("-d", "--digits", required=True,
37                          help="directory path for digit and arrow images")
38      parser.add_argument("-w", "--work", required=True,
39                          help="directory path for work images")
40      parser.add_argument("-W", "--work-save", action="store_true",
41                          help="flag if intermediate images " +
42                          "are to be saved to work directory")
43
44      args = vars(parser.parse_args())
45
46      dir_input = args["input"]
47
48      img_files = [os.path.join(dir_input, f)
49                   for f in os.listdir(dir_input)
50                   if os.path.isfile(os.path.join(dir_input, f))
51                   and os.path.splitext(f)[1] in {".jpg", ".png"}]
52
53      return args, img_files
54
55
56  def parse_image_file(img_file):
57      """
58      Parse an image filepath into rootname, extension, and the integer ID.
59
60      Parameters
61      ----------
62      img_file : string
63
64      Returns
65      -------
66      file_root : string
67          Root of the basename of `img_file`.
68      file_ext : string
69          Extension of the basename of `img_file`.
70      file_id : string
71          Integer ID of `img_file`.
72
73      """
74      file_root, file_ext = os.path.splitext(os.path.basename(img_file))
75      match_id = re.search("[0-9]+", file_root)
76      file_id = match_id.group(0) if match_id else ""
77      return (file_root, file_ext, file_id)
```

A.6 task_1.py

```
#!/usr/bin/env python3

import numpy as np
import cv2
from timeit import default_timer as timer

from parser import *
from region import *
from chain import *
from knn import *


# task 1
args, img_files = parse_input()

# build classifier
print(f"> building classifier")
knn_digits = build_knn_digits(args["digits"], 5, 7)

# locate and classify the digits of each building sign
for img_file in img_files:
    time_img = timer()
    def timing():
        return f"{timer() - time_img:>.1f} s>"

    file_root, file_ext, file_id = parse_image_file(img_file)
    print(f"{img_file} -> ({file_root}, {file_ext}, {file_id})")

    def write_image_to_work(suffix, img_work):
        cv2.imwrite(f"{args['work']}/{file_root}_{suffix}{file_ext}", img_work)
        return

    print(f"{timing()} reading {img_file}")
    img = cv2.imread(img_file, cv2.IMREAD_COLOR)
    if img is None:
        print(f"{img_file} could not be opened")
        continue
    H, W = img.shape[:2]

    if args["work_save"]:
        print(f"{timing()} writing image to work")
        write_image_to_work("0", img)

    print(f"{timing()} converting to grayscale")
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    if args["work_save"]:
        print(f"{timing()} writing grayscale image to work")
        write_image_to_work("1", img_gray)

    print(f"{timing()} calculating MSER")
    mser = cv2.MSER_create()
    mser.setMinArea(45)
    mser.setMaxArea(2000)
    mser.setDelta(20)
    point_sets, boxes = mser.detectRegions(img_gray)
```

```
57
58      print(f"{timing()} constructing regions")
59      regions = [Region(ps) for ps in point_sets]
60
61      if args["work_save"]:
62          print(f"{timing()} writing regions ({len(regions)})")
63          write_image_to_work("2_0", draw_regions(regions, (H, W)))
64
65      print(f"{timing()} removing overlapping regions")
66      regions = remove_overlapping(regions, max_overlap=0.8)
67
68      if args["work_save"]:
69          print(f"{timing()} writing regions ({len(regions)})")
70          write_image_to_work("2_1", draw_regions(regions, (H, W)))
71
72      print(f"{timing()} filtering regions by aspect ratio")
73      # 0.8 for directions, 1.2 for digits
74      regions = list(filter(lambda r: 1.2 <= r.box.aspect <= 3.0, regions))
75
76      if args["work_save"]:
77          print(f"{timing()} writing regions ({len(regions)})")
78          write_image_to_work("2_2", draw_regions(regions, (H, W)))
79
80      print(f"{timing()} removing occluded hole regions")
81      regions = remove_occluded_holes(regions, max_boundary_distance=10)
82
83      if args["work_save"]:
84          print(f"{timing()} writing regions ({len(regions)})")
85          write_image_to_work("2_3", draw_regions(regions, (H, W)))
86
87      print(f"{timing()} removing highly filled regions")
88      regions = list(filter(lambda r: r.fill <= 0.85, regions))
89
90      if args["work_save"]:
91          print(f"{timing()} writing regions ({len(regions)})")
92          write_image_to_work("2_4", draw_regions(regions, (H, W)))
93
94      print(f"{timing()} calculating chains of similar, adjacent regions")
95      chains = find_chains(regions)
96
97      if args["work_save"]:
98          print(f"{timing()} writing chains ({len(chains)})")
99          img_chains = draw_regions(regions, (H, W))
100         for chain in chains:
101             chain_box = covering_box([r.box for r in chain])
102             cv2.rectangle(
103                 img_chains, chain_box.tl, chain_box.br, (255, 255, 255), 1)
104         write_image_to_work("3", img_chains)
105
106     print(f"{timing()} filtering chains by length")
107     chains = list(filter(lambda c: len(c) <= 3, chains))
108
109     if not chains:
110         print(f"{timing()} no suitable chains found")
111         print(f"{timing()} ")
112         continue
113
114     if args["work_save"]:
```

```
115         print(f"{timing()} writing regions of interest")
116         rois = [covering_box([r.box for r in c]) for c in chains]
117         rois = merge_overlapping(rois, max_overlap=0.01)
118 
119         img_rois = img.copy()
120         for i, roi in enumerate(rois):
121             cv2.rectangle(img_rois, roi.tl, roi.br, (255, 255, 255), 1)
122         write_image_to_work("4", img_rois)
123 
124         for i, roi in enumerate(rois):
125             img_roi = img[roi.indexes]
126             write_image_to_work(f"4_{i}", img_roi)
127 
128     print(f"{timing()} selecting chain most likely to be digits")
129     chain_digits = cluster_largest_otsu_separations(img, chains)[0]
130 
131     if args["work_save"]:
132         print(f"{timing()} writing digit chain")
133         img_digits = img[covering_box([r.box for r in chain_digits]).indexes]
134         write_image_to_work("5", img_digits)
135 
136     print(f"{timing()} classifying digits")
137     features_digits = np.array(
138         [np.ravel(r.spatial_occupancy(5, 7))
139          for r in chain_digits])
140     predicted_digits = knn_digits.predict(features_digits, k=3)
141 
142     print(f"{timing()} writing output for {file_root}{file_ext}")
143     img_digits = img[covering_box([r.box for r in chain_digits]).indexes]
144     cv2.imwrite(f"{args['output']}/DetectedArea{file_id}{file_ext}", img_digits)
145 
146     with open(f"{args['output']}/Building{file_id}.txt", "w") as out_file:
147         str_digits = "".join(map(str, predicted_digits))
148         print(f"Building {str_digits}", file=out_file)
149 
150     print(f"{timing()} ")
```

A.7 task_2.py

```
1  #!/usr/bin/env python3
2  
3  import numpy as np
4  import cv2
5  from timeit import default_timer as timer
6  
7  from parser import *
8  from region import *
9  from chain import *
10 from knn import *
11 
12 
13 # task 2
14 args, img_files = parse_input()
15 
16 # build classifiers
17 print(f"> building classifiers")
```

```
18 knn_digits = build_knn_digits(args["digits"], 3, 5)
19 knn_arrows = build_knn_arrows(args["digits"], 2, 2)
20
21 # locate and classify the digits of each line of each directional sign
22 for img_file in img_files:
23     time_img = timer()
24     def timing():
25         return f"{timer() - time_img:>.1f} s>"
26
27     file_root, file_ext, file_id = parse_image_file(img_file)
28     print(f"{img_file} -> ({file_root}, {file_ext}, {file_id})")
29
30     def write_image_to_work(suffix, img_work):
31         cv2.imwrite(f"{args['work']}/{file_root}_{suffix}{file_ext}", img_work)
32         return
33
34     print(f"{timing()} reading {img_file}")
35     img = cv2.imread(img_file, cv2.IMREAD_COLOR)
36     if img is None:
37         print(f"{img_file} could not be opened")
38         continue
39     H, W = img.shape[:2]
40
41     if args["work_save"]:
42         print(f"{timing()} writing image to work")
43         write_image_to_work("0", img)
44
45     print(f"{timing()} converting to grayscale")
46     img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
47
48     if args["work_save"]:
49         print(f"{timing()} writing grayscale image to work")
50         write_image_to_work("1", img_gray)
51
52     print(f"{timing()} calculating MSER")
53     mser = cv2.MSER_create()
54     mser.setMinArea(25)
55     mser.setMaxArea(2000)
56     mser.setDelta(20)
57     point_sets, boxes = mser.detectRegions(img_gray)
58
59     print(f"{timing()} constructing regions")
60     regions = [Region(ps) for ps in point_sets]
61
62     if args["work_save"]:
63         print(f"{timing()} writing regions ({len(regions)})")
64         write_image_to_work("2_0", draw_regions(regions, (H, W)))
65
66     print(f"{timing()} removing overlapping regions")
67     regions = remove_overlapping(regions, max_overlap=0.8)
68
69     if args["work_save"]:
70         print(f"{timing()} writing regions ({len(regions)})")
71         write_image_to_work("2_1", draw_regions(regions, (H, W)))
72
73     print(f"{timing()} filtering regions by aspect ratio")
74     regions = list(filter(lambda r: 0.75 <= r.box.aspect <= 3.0, regions))
75
```

```
76      if args["work_save"]:
77          print(f"{timing()} writing regions ({len(regions)})")
78          write_image_to_work("2_2", draw_regions(regions, (H, W)))
79
80      print(f"{timing()} removing occluded hole regions")
81      regions = remove_occluded_holes(regions, max_boundary_distance=10)
82
83      if args["work_save"]:
84          print(f"{timing()} writing regions ({len(regions)})")
85          write_image_to_work("2_3", draw_regions(regions, (H, W)))
86
87      print(f"{timing()} removing highly filled regions")
88      regions = list(filter(lambda r: r.fill <= 0.85, regions))
89
90      if args["work_save"]:
91          print(f"{timing()} writing regions ({len(regions)})")
92          write_image_to_work("2_4", draw_regions(regions, (H, W)))
93
94      print(f"{timing()} finding chains of similar, adjacent regions")
95      chains = find_chains(regions)
96
97      if args["work_save"]:
98          print(f"{timing()} writing chains ({len(chains)})")
99          img_chains = draw_regions(regions, (H, W))
100         for chain in chains:
101             chain_box = covering_box([r.box for r in chain])
102             cv2.rectangle(
103                 img_chains, chain_box.tl, chain_box.br, (255, 255, 255), 1)
104         write_image_to_work("3", img_chains)
105
106     print(f"{timing()} filtering chains by length")
107     chains = list(filter(lambda c: len(c) <= 3, chains))
108
109     if not chains:
110         print(f"{timing()} no suitable chains found")
111         print(f"{timing()} ")
112         continue
113
114     if args["work_save"]:
115         print(f"{timing()} writing regions of interest")
116         rois = [covering_box([r.box for r in c]) for c in chains]
117         rois = merge_overlapping(rois, max_overlap=0.01)
118
119         img_rois = img.copy()
120         for i, roi in enumerate(rois):
121             cv2.rectangle(img_rois, roi.tl, roi.br, (255, 255, 255), 1)
122         write_image_to_work("4", img_rois)
123
124         for i, roi in enumerate(rois):
125             img_roi = img[roi.indexes]
126             write_image_to_work(f"4_{i}", img_roi)
127
128     print(f"{timing()} finding aligned chains")
129     aligned_chains = find_aligned_chains(chains)
130
131     if not aligned_chains:
132         print(f"{timing()} no suitable aligned chains found")
133         print(f"{timing()} ")
```

```
134         continue
135
136     if args["work_save"]:
137         print(f"{timing()} writing aligned chains")
138         chain_boxes = [covering_box([r.box for r in c]) for c in aligned_chains]
139
140         img_ac = img.copy()
141         for box in chain_boxes:
142             cv2.rectangle(img_ac, box.tl, box.br, (255, 255, 255), 1)
143
144         ac_box = covering_box(chain_boxes)
145         cv2.rectangle(img_ac, ac_box.tl, ac_box.br, (255, 255, 255), 2)
146         write_image_to_work("5", img_ac)
147
148     print(f"{timing()} finding any missing digits")
149     aligned_chains = find_missing_digits(aligned_chains, img_gray)
150
151     if args["work_save"]:
152         print(f"{timing()} writing aligned chains (with found digits)")
153         chain_boxes = [covering_box([r.box for r in c]) for c in aligned_chains]
154
155         img_ac = img.copy()
156         for box in chain_boxes:
157             cv2.rectangle(img_ac, box.tl, box.br, (255, 255, 255), 1)
158
159         ac_box = covering_box(chain_boxes)
160         cv2.rectangle(img_ac, ac_box.tl, ac_box.br, (255, 255, 255), 2)
161         write_image_to_work("6", img_ac)
162
163     print(f"{timing()} finding each chain's associated arrow")
164     aligned_chains_arrows = find_arrows(aligned_chains, regions)
165
166     if args["work_save"]:
167         print(f"{timing()} writing aligned chains with arrows")
168         chain_arrow_boxes = [covering_box([r.box for r in c] + [a.box])
169                              for c, a in aligned_chains_arrows]
170
171         img_aca = img.copy()
172         for box in chain_arrow_boxes:
173             cv2.rectangle(img_aca, box.tl, box.br, (255, 255, 255), 1)
174
175         aca_box = covering_box(chain_arrow_boxes)
176         cv2.rectangle(img_aca, aca_box.tl, aca_box.br, (255, 255, 255), 2)
177         write_image_to_work("7", img_aca)
178
179         for i, (c, a) in enumerate(aligned_chains_arrows):
180             write_image_to_work(f"7_{i}", draw_regions(c + [a]))
181
182     print(f"{timing()} classifying digits and arrows")
183     predicted = []
184     for chain_digits, arrow in aligned_chains_arrows:
185
186         features_digits = np.array(
187             [np.ravel(r.spatial_occupancy(3, 5))
188              for r in chain_digits])
189         predicted_digits = knn_digits.predict(features_digits, k=3)
190
191         features_arrow = np.array([np.ravel(arrow.spatial_occupancy(2, 2))])
```

```
192             predicted_arrow = knn_arrows.predict(features_arrow, k=3)
193             predicted.append((predicted_digits, predicted_arrow))
194
195         print(f"{timing()} writing output for {file_root}{file_ext}")
196         aca_box = covering_box(
197             [covering_box([r.box for r in c] + [a.box])
198              for c, a in aligned_chains_arrows])
199         img_sign = img[aca_box.indexes]
200         cv2.imwrite(f"{args['output']}/DetectedArea{file_id}{file_ext}", img_sign)
201
202         with open(f"{args['output']}/BuildingList{file_id}.txt", "w") as out_file:
203             for ds, a in predicted:
204                 str_digits = "".join(map(str, ds))
205                 if a[0] == "L":
206                     str_arrow = "to the left"
207                 else:
208                     str_arrow = "to the right"
209
210                 print(f"Building {str_digits} {str_arrow}", file=out_file)
211
212         print(f"{timing()} ")
```