Machine Perception Assignment 2  
House Number Extraction

Reece Jones

Abstract

This report presents a complete pipeline for extracting and recognising house numbers from constrained real-world images. The pipeline utilises maximally stable extremal regions (MSER), hand-crafted region filtering and text path extraction logic, and support vector machine (SVM) digit classification. Perfect accuracy is achieved on the provided training and validation images.

1. Introduction

The problem of extracting text from real-world images is difficult, and likely remains an open problem. Several existing methods were examined, [1], [2], [3], and [4], all of which utilise large-scale machine learning in some manner to deal with the variations present in natural scenes (lighting, affine transformations, perspective, etc.). A particularly difficult element of this problem is the extraction of the characters, typically requiring machine learning to classify character and noncharacter regions. Fortunately, as large-scale machine learning is not feasible for this task, the images provided are constrained enough such that hand-crafted algorithms can achieve good performance. Primarily:

* Digit fonts are simple and limited,
* Digits are mostly uniform, bright colours on uniform, dark backgrounds,
* Numbers are near to horizontal,
* Limited perspective warping is present,
* Images contain at most one house number each, and
* Any other text present is small.

Inspiration was taken from [1], as the methodology was similar to my own thoughts on the task and could be adjusted to require less machine learning.

1. Pipeline Methodology

The developed pipeline consists of three main sections: digit region detection, region filtering, house number extraction, and digit classification.

A. Digit Region Detection

Multiple techniques exist to detect regions in which text may be present. This is not necessarily a trivial problem. A simple threshold or Otsu’s binarisation is unlikely to be robust, as no guarantees can be made about the nondigit regions of the image – these approaches are too global. Adaptive thresholding, image gradients, edges, and contours were considered, but the results were difficult to utilise and were not reliable. One solution used by [2] and [3] is a sliding-window approach, where the entire image is mapped to a surface which represents the likelihood of text being present. However, this approach requires evaluation of arbitrary image patches for containment or noncontainment of text, likely requiring a machine learning model for which we do not have sufficient data nor computational resources. Attempts without machine learning (i.e. hand-crafted features) were too slow in Python to be feasible. After running out of ideas, the approach from [1] was used: assume each digit is a connected extremal region and use MSER [5] blob detection. MSER is computationally efficient and succeeds in detecting the digits in all the provided images.

1. Region Filtering and Adjustment

MSER detects many regions which are not digits, so some manner of filtering is required to remove nondigit regions. An initial idea was to use the digit classifier to ignore regions that were not confidently classified, but SVMs do not really support this functionality. Another approach, used by [1], is to train a separate classifier for discriminating between digit and nondigit regions, but is, as explained in the previous paragraph, not feasible for this task. Therefore, a series of hand-crafted filters is implemented, based on observations of the provided images.



Figure 1. Bounding boxes of MSERs detected in training image tr15.jpg.

The first filter is based on regions’ bounding box areas. Any regions whose bounding box area is less than 50 pixels in area is discarded, deemed too small to contain a recognisable digit. Any regions whose bounding box is greater than 25% of the total image are is discarded, as the digits in the images provided are typically not this large. This

The second filter is based on the regions’ bounding box aspect ratios. Digits are typically taller than they are wide, so if the aspect ratio is greater than 0.8, the region is discarded. Digits are typically not extremely thin, though, so if the aspect ratio is less than 0.2, the region is discarded also. This filter is a simple and efficient method for removing a large proportion of undesired detected regions.



Figure 2. Bounding boxes of filtered regions from training image tr15.jpg.

The third filter considers the pixel intensity of the regions. One of the constraints on the provided images is that the digits are a light colour on a dark background, i.e. the digits are bright compared to their local area (this fact is also relied on for processing further in the pipeline). However, MSER produces regions which are both locally bright and dark. To filter out the locally dark regions, a region’s pixels are compared to the foreground pixels determined by Otsu’s binarisation of the image patch within the region bounding box. If less than 75% of the region’s pixels are foreground, the region is discarded.

The fourth filter considers the connected components within the regions’ bounding boxes. Digits consist of a single foreground component, but MSER may yield regions whose bounding boxes span multiple foreground components. To filter such regions out, the image patch within the region bounding box is binarised with Otsu’s binarisation, the connected components are found, and the areas of the components examined. If there is more than one component with an area greater than 5% of the region bounding box, the region is discarded.

The fifth filter also examines the connected components within the regions’ bounding boxes. Digits are typically not connected to any other foreground regions, yet MSER’s regions are sometimes subregions of the local foreground. This filter attempts to expand the bounding box of a region by a few pixels until the border pixels do not contain any of the largest foreground component. If this can be done, the original region’s box is replaced with the expanded box. If this cannot be done, the region is discarded.

As a final step, duplicate regions are removed. MSER and/or filter five often produce regions which are very nearly the same (e.g. bounds differing by a few pixels). Any regions which whose centres are within a few pixels of each other are grouped as equivalent. Out of each equivalency group, the region whose bounding box is the smallest is selected, based on the assumption that the smallest box will be the tightest fit for the foreground area.

After filtering, typically only regions containing characters remain, sometimes along with a few spurious regions. An example is shown in figures 1 (before filtering) and 2 (after filtering). Note that I do not believe it is possible to distinguish between digits and characters without some form of machine learning.

1. House Number Extraction

Since it was not possible to filter out all nondigit regions, further processing is required to isolate the regions constituting the house number. This step begins by identifying plausible text paths, then selecting the path most likely to be the house number.

The text path identification is inspired by [1]. For each region bounding box *B*0, the formation of a text path *P* beginning at *B0* and consisting of regions bounding boxes *B0*, *B1*, *B2*, etc. is attempted. A region with bounding box *Bx* is added to *P* if all of the following are true:

1. The angle from the centre of *B0* to the centre of *Bx* is in the range [-20°, 20°].
2. The distance between the centre of *Bx* and the centre of *Bx-1* in the X-axis is less than three times the maximum width of all boxes currently in *P*. (As done in [1].)
3. The absolute difference in height between *B*x and Bx-1 is less than or equal to 25% the height of *Bx*.

Condition 1 is derived from the observation that house numbers are roughly horizontal in the images. Condition 2 is derived from the observation that digits in the same number are spaced near to each other in the X-axis. Condition 3 is derived from the assumption that digits in the same number are roughly the same height, and that perspective warping present in the images is minimal.  
Regions are considered in the order of their X position, and the path is terminated if either of conditions 2 or 3 are violated. In this manner, plausible text paths are formed, extending to the right as far as possible.

Once text paths are formulated, the one containing the house number must be selected. Under the assumption that the house numbers contain the tallest characters, any paths with mean box height less than 75% of the largest mean box height are discarded. Of the remaining paths, the one containing the most regions is chosen.

1. Digit Recognition

The final step in the pipeline is classification of the digits constituting the house number.

To generate the feature descriptor used for classification of a digit image patch, the patch is first taken as greyscale. Colour information is deemed unnecessary, since it has already been established that the digit differs in intensity from the local background. To isolate the digit, the patch is cropped to the foreground and the background is removed with Otsu’s binarisation. The patch is then resized to a consistent size of 32x32 pixels (at this stage in the pipeline, the aspect ratio of the digit need not be preserved, as it is not a distinguishing factor). Initially, the resulting raw pixel data was used as the feature descriptor, but this was found to be sensitive to small variations in the digit position. Histogram of oriented gradients (HOG) [6] was instead chosen in the belief it would focus more on the shape of the digit.

An SVM is used as a classifier, as they are known to be effective for digit recognition [1, 3, 4], and are simple to implement in OpenCV. Neural networks, particularly convolutional neural networks, could also be effective, however no neural network libraries are available in the provided virtual machine environment, and would likely require more training data than was available. The SVM is trained on the provided labelled digit images, using OpenCV’s automatic training to optimise the SVM parameters via cross validation. Linear and radial basis function kernels were tested; they were found to be roughly equally effective, however the training time for the linear kernel was significantly less.

1. Experimental Results
2. Digit Classifier

The performance of the digit classifier was evaluated using the provided training data, since no other data was available, using cross validation. 100% accuracy was consistently achieved on both the training and test data. While the performance is good, ideally a larger dataset would be utilised to determine the generalisation power of the model more extensively.

1. Complete Pipeline

The performance of the pipeline was evaluated on the provided training and validation images. The accuracy of the recognised house numbers was 100% for both image sets. The bounding boxes of the extracted house numbers are more difficult to validate, but visual inspection shows acceptable results – assumedly, the bounding boxes are close enough to a ground truth to produce good house number accuracy.

1. Discussion and Limitations

The pipeline achieves excellent performance on the training and validation datasets, but it is probable this is due to over-specialisation, specifically in the region filtering. While the filtering components may be acceptable in theory, in practice they rely on several arbitrary values and thresholds. These values were, of course, selected to produce the desired results for the training images, and as such no guarantees about their generalisation to other images is given. Additionally, the nature of the filters as hard cut-offs based on a single region feature likely reduces the robustness of the pipeline. Similarly, in the house number extraction algorithm, no trade-offs are considered between the text path features, simply the path with the tallest and most boxes are chosen. An approach which instead considers multiple region/path features in combination and selects the regions/paths which are globally most likely to be relevant, would be more robust. However, such an approach would be only be feasible with the use of machine learning, for which in this task there was likely not enough data available.

References

[1] L. Neumann and J. Matas, “A method for text localization and recognition in real-world images,” *Computer Vision – ACCV 2010*, 2010, doi: 10.1007/978-3-642-19318-7\_60.

[2] Q. Guo, F. Wang, J. Lei, D. Tu and G. Li, “Convolutional feature learning and hybrid CNN-HMM for scene number recognition,” *Neurocomputing*, vol. 184, pp. 78-90, Apr. 2016, doi: 10.1016/j.neucom.2015.07.135.

[3] A. Coates et al., “Text detection and character recognition in scene images with unsupervised feature learning,” *2011 International Conference on Document Analysis and Recognition*, Beijing, pp. 440-445, 2011, doi: 10.1109/ICDAR.2011.95.

[4] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Ng, “Reading digits in natural images with unsupervised feature learning,” *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011, http://ufldl.stanford.edu/housenumbers/nips2011\_housenumbers.pdf.

[5] J. Matas, O. Chum, M. Urban and T. Pajdla, “Robust wide-baseline stereo from maximally stable extremal regions,” *Image and Vision Computing*, vol. 22, no. 10, pp. 761-767, 2004, doi: 10.1016/j.imavis.2004.02.006.

[6] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, San Diego, CA, USA, pp. 886-893 vol. 1, 2005, doi: 10.1109/CVPR.2005.177.

Pipeline Source Code

src/pipeline/digit\_descriptor.py

*# Defines the descriptor used for digit recognition.*import cv2 as cv  
import numpy  
  
  
\_\_all\_\_ = [  
 **'digit\_descriptor'**]  
  
  
*# Produces a feature descriptor for an digit image that is used for recognition.*def digit\_descriptor(image\_grey: numpy.ndarray) -> numpy.ndarray:  
 *# Crop the image to just the foreground area.* \_, binary = cv.threshold(image\_grey, 0, 255, cv.THRESH\_OTSU)  
 x, y, w, h = cv.boundingRect(binary)  
 image\_grey = image\_grey[y:y + h, x:x + w]  
  
 *# Resize to uniform size.* image\_grey = cv.resize(image\_grey, (32, 32), interpolation=cv.INTER\_CUBIC)  
 image\_grey = cv.normalize(image\_grey, None, 0, 255, cv.NORM\_MINMAX)  
 *# Threshold to remove background. Retain foreground for better representation of the digit.* \_, image\_grey = cv.threshold(image\_grey, 0, 255, cv.THRESH\_OTSU + cv.THRESH\_TOZERO)  
 *# Use HOG descriptor, which is better at handling variations in the image than raw image data.* descriptor = \_hog.compute(image\_grey).ravel()  
 return descriptor  
  
  
\_hog = cv.HOGDescriptor((32, 32), (16, 16), (8, 8), (8, 8), 9)

src/pipeline/number\_extract.py

*# Functions for extracting the digits and house numbers.*import cv2 as cv  
from itertools import product  
from math import atan2, hypot, radians  
import numpy  
from typing import Sequence, Tuple  
  
  
\_\_all\_\_ = [  
 **'detect\_regions'**,  
 **'filter\_regions'**,  
 **'select\_number'**]  
  
  
BoundingBox = Tuple[float, float, float, float]  
Point = Tuple[float, float]  
Line = Tuple[Point, Point]  
  
  
*# Finds MSERS.*def detect\_regions(image\_grey: numpy.ndarray):  
 max\_area = int(image\_grey.shape[0] \* image\_grey.shape[1] \* REGION\_AREA\_MAX)  
 mser = cv.MSER\_create(MSER\_DELTA, REGION\_AREA\_MIN, max\_area)  
 msers, boxes = mser.detectRegions(image\_grey)  
 return boxes, msers  
  
  
*# Finds regions that are possibly digits.*def filter\_regions(image\_grey: numpy.ndarray, region\_boxes: Sequence[BoundingBox],  
 region\_points: numpy.ndarray) -> Sequence[BoundingBox]:  
 max\_area = int(image\_grey.shape[0] \* image\_grey.shape[1] \* REGION\_AREA\_MAX)  
  
 adjusted\_boxes = []  
 for box, points in zip(region\_boxes, region\_points):  
 x, y, w, h = box  
  
 *# OpenCV's MSER area bounds don't seem to always work.* if not REGION\_AREA\_MIN <= w \* h <= max\_area:  
 continue  
  
 *# Filter out regions with aspect ratios unlikely to occur for digits.* if not REGION\_ASPECT\_RATIO\_RANGE[0] <= w / h <= REGION\_ASPECT\_RATIO\_RANGE[1]:  
 continue  
  
 region\_grey = image\_grey[y:y + h, x:x + w]  
  
 *# Further processing and recognition assumes that the digit is the foreground (i.e. light on  
 # a dark background) in the local area. But MSER can produce regions that are dark on a  
 # light background, so need to filter out those.* if not \_region\_is\_foreground(region\_grey, box, points):  
 continue  
  
 *# Digits typically only have 1 main foreground component.* if \_has\_multiple\_components(region\_grey):  
 continue  
  
 *# Filters out regions whose foreground is not isolated.* is\_isolated, box = \_is\_foreground\_isolated(image\_grey, box)  
 if not is\_isolated:  
 continue  
  
 adjusted\_boxes.append(box)  
  
 *# Remove boxes are are basically the same.* adjusted\_boxes = \_remove\_equivalent\_boxes(adjusted\_boxes)  
  
 return adjusted\_boxes  
  
  
*# Selects the regions (from detect\_regions()) that form the house number.*def select\_number(boxes: Sequence[BoundingBox]) -> Sequence[BoundingBox]:  
 if len(boxes) <= 1:  
 return boxes  
  
 boxes = sorted(boxes, key=lambda b: b[0])  
  
 paths = []  
 for i, box1 in enumerate(boxes):  
 p = [box1]  
 c1x = box1[0] + box1[2] / 2  
 c1y = box1[1] + box1[3] / 2  
 for j in range(i + 1, len(boxes)):  
 box2 = boxes[j]  
 c2x = box2[0] + box2[2] / 2  
 c2y = box2[1] + box2[3] / 2  
 *# Create only roughly horizontal paths.* if abs(atan2(c2y - c1y, c2x - c1x)) <= NUMBER\_PATH\_ANGLE\_THRESHOLD:  
 *# Select only boxes that are near to the previous box in the X axis.* if c2x - (p[-1][0] + p[-1][2] / 2) <= 3 \* max(b[2] for b in p):  
 *# Select only boxes that have similar height to the previous box.* if abs(p[-1][3] - box2[3]) / box2[3] <= NUMBER\_PATH\_HEIGHT\_DIFF\_THRESHOLD:  
 p.append(box2)  
 else:  
 break  
 else:  
 break  
 paths.append(p)  
 path\_lengths = [len(p) for p in paths]  
 box\_heights = [numpy.array([b[3] for b in p]) for p in paths]  
 box\_height\_means = numpy.array([numpy.mean(hs) for hs in box\_heights])  
  
 path\_filter = numpy.zeros(len(paths), numpy.bool)  
 *# Consider only the paths with the tallest boxes.* path\_filter[box\_height\_means / numpy.max(box\_height\_means) < NUMBER\_TALL\_THRESHOLD] = True  
 *# Then select the longest path.* best\_path = numpy.ma.argmax(numpy.ma.masked\_array(path\_lengths, path\_filter))  
 return paths[best\_path]  
  
  
*# Checks if a significant amount of pixels in a region are foreground when considering the local area.*def \_region\_is\_foreground(region\_grey: numpy.ndarray, box: BoundingBox, points: numpy.ndarray) -> bool:  
 \_, binary = cv.threshold(region\_grey, 0, 1, cv.THRESH\_OTSU)  
 indices = numpy.flip(points - [box[0], box[1]], 1)  
 fg\_points = binary.ravel()[numpy.ravel\_multi\_index((indices[:, 0], indices[:, 1]), binary.shape)]  
 region\_fg\_ratio = numpy.average(fg\_points)  
 return region\_fg\_ratio >= REGION\_FG\_RATIO\_THRESHOLD  
  
  
*# Checks if there are multiple (significant) foreground components.*def \_has\_multiple\_components(region\_grey: numpy.ndarray) -> bool:  
 \_, binary = cv.threshold(region\_grey, 0, 255, cv.THRESH\_OTSU)  
 \_, labels = cv.connectedComponents(binary)  
 fg\_component\_areas = numpy.bincount(labels.ravel())[1:]  
 max\_area = numpy.max(fg\_component\_areas)  
 significant\_components = fg\_component\_areas / max\_area > REGION\_SIGNIFICANT\_COMPONENT\_AREA\_THRESHOLD  
 return numpy.count\_nonzero(significant\_components) > 1  
  
  
*# Checks if a region's foreground is not part of a larger object.*def \_is\_foreground\_isolated(image\_grey: numpy.ndarray, box: BoundingBox) -> Tuple[bool, BoundingBox]:  
 x1, y1, w, h = box  
 x2 = x1 + w  
 y2 = y1 + h  
 is\_isolated = True  
 *# Try to grow region side by side until it covers entire foreground area.* for axis, direction in product(((1, 0), (0, 1)), (-1, 1)):  
 for \_ in range(0, REGION\_GROWTH + 1):  
 prev\_x1 = x1  
 prev\_y1 = y1  
 prev\_x2 = x2  
 prev\_y2 = y2  
 if direction < 0:  
 x1 = max(x1 - axis[0], 0)  
 y1 = max(y1 - axis[1], 0)  
 elif direction > 0:  
 x2 = min(x2 + axis[0], image\_grey.shape[1])  
 y2 = min(y2 + axis[1], image\_grey.shape[0])  
  
 region\_grey = image\_grey[y1:y2, x1:x2]  
  
 \_, binary = cv.threshold(region\_grey, 0, 1, cv.THRESH\_OTSU)  
 *# Find the label of the main foreground component.* \_, labels = cv.connectedComponents(binary)  
 fg\_component\_areas = numpy.bincount(labels.ravel())[1:]  
 fg\_label = numpy.argmax(fg\_component\_areas) + 1  
  
 *# Check if the main foreground component touches the border of the box.* border\_mask = numpy.zeros\_like(region\_grey, numpy.bool)  
 if direction < 0 and axis[0]:  
 border\_mask[:, 0] = True  
 elif direction > 0 and axis[0]:  
 border\_mask[:, -1] = True  
 elif direction < 0 and axis[1]:  
 border\_mask[0, :] = True  
 elif direction > 0 and axis[1]:  
 border\_mask[-1, :] = True  
 border\_labels = labels[border\_mask]  
 if numpy.all(border\_labels != fg\_label):  
 *# Isolated in this axis and direction.  
 # Revert box so we don't get a 1 pixel gap around the foreground.* x1 = prev\_x1  
 y1 = prev\_y1  
 x2 = prev\_x2  
 y2 = prev\_y2  
 break  
 else:  
 is\_isolated = False  
 break  
 return is\_isolated, (x1, y1, x2 - x1, y2 - y1)  
  
  
*# Find all the groups of boxes that are effectively the same, and from those groups chooses only  
# the regions with minimal area.*def \_remove\_equivalent\_boxes(boxes: Sequence[BoundingBox]) -> Sequence[BoundingBox]:  
 boxes = list(set(boxes))  
 centres = [(x + w / 2, y + h / 2) for x, y, w, h in boxes]  
 equivalencies = []  
 for i, (box1, (c1x, c1y)) in enumerate(zip(boxes, centres)):  
 tmp = {box1}  
 for j, (box2, (c2x, c2y)) in enumerate(zip(boxes, centres)):  
 if box1 is not box2:  
 *# Consider boxes to be the same if their centres are close.* distance = hypot(c1x - c2x, c1y - c2y)  
 if distance < EQUIVALENT\_BOX\_DISTANCE:  
 tmp.add(box2)  
 equivalencies.append(tmp)  
 for \_ in equivalencies:  
 for e1 in equivalencies:  
 for e2 in equivalencies:  
 if e1 != e2:  
 if any(b in e1 for b in e2):  
 for b in e2:  
 e1.add(b)  
 boxes = list(set(min(e, key=lambda b: b[2] \* b[3]) for e in equivalencies))  
 return boxes  
  
  
MSER\_DELTA = 15  
REGION\_AREA\_MIN = 50 *# Pixels*REGION\_AREA\_MAX = 0.5 \* 0.5 *# Fraction of image area*REGION\_ASPECT\_RATIO\_RANGE = (0.2, 0.8)  
REGION\_FG\_RATIO\_THRESHOLD = 0.75  
REGION\_SIGNIFICANT\_COMPONENT\_AREA\_THRESHOLD = 0.05 *# Fraction of largest component's area*REGION\_GROWTH = 5 *# Pixels*EQUIVALENT\_BOX\_DISTANCE = 5 *# Pixels*NUMBER\_PATH\_ANGLE\_THRESHOLD = radians(20)  
NUMBER\_PATH\_HEIGHT\_DIFF\_THRESHOLD = 0.25 *# Fraction of current box*NUMBER\_TALL\_THRESHOLD = 0.75 *# Fraction of tallest boxes' height*

src/pipeline/pipeline.py

from .digit\_descriptor import digit\_descriptor  
from .number\_extract import detect\_regions, filter\_regions, select\_number  
from .recognition\_model import DigitRecognitionModel  
import numpy  
from typing import Sequence, Tuple  
  
  
\_\_all\_\_ = [  
 **'HouseNumberRecognitionPipeline'**]  
  
  
BoundingBox = Tuple[float, float, float, float]  
  
  
class HouseNumberRecognitionPipeline:  
 class Result:  
 def \_\_init\_\_(self, raw\_regions: Sequence[BoundingBox],  
 filtered\_regions: Sequence[BoundingBox],  
 house\_number\_regions: Sequence[BoundingBox],  
 house\_number: Sequence[int]) -> None:  
 self.raw\_regions = raw\_regions  
 self.filtered\_regions = filtered\_regions  
 self.house\_number\_regions = house\_number\_regions  
 self.house\_number = house\_number  
  
 def \_\_init\_\_(self, recognition\_model\_file) -> None:  
 self.recognition\_model = DigitRecognitionModel(recognition\_model\_file)  
  
 def process(self, image\_grey: numpy.ndarray) -> Result:  
 raw\_boxes, regions = detect\_regions(image\_grey)  
 filtered\_boxes = filter\_regions(image\_grey, raw\_boxes, regions)  
 house\_number\_boxes = select\_number(filtered\_boxes)  
 house\_number = []  
 for x, y, w, h in house\_number\_boxes:  
 region = image\_grey[y:y + h, x:x + w]  
 descriptor = digit\_descriptor(region)  
 digit = self.recognition\_model.predict\_single(descriptor)  
 house\_number.append(digit)  
 return self.Result(raw\_boxes, filtered\_boxes, house\_number\_boxes, house\_number)

src/pipeline/recognition\_model.py

*# Defines the SVM used to recognise digits.*import cv2 as cv  
import numpy  
  
  
\_\_all\_\_ = [  
 **'DigitRecognitionModel'**]  
  
  
class DigitRecognitionModel:  
 def \_\_init\_\_(self, file\_path=None) -> None:  
 if file\_path is None:  
 self.svm = cv.ml.SVM\_create()  
 self.svm.setKernel(cv.ml.SVM\_LINEAR)  
 self.svm.setType(cv.ml.SVM\_C\_SVC)  
 else:  
 self.svm = cv.ml.SVM\_load(file\_path)  
  
 def train(self, inputs: numpy.ndarray, labels: numpy.ndarray) -> None:  
 self.svm.trainAuto(inputs, cv.ml.ROW\_SAMPLE, labels, 4)  
  
 def predict\_single(self, input: numpy.ndarray) -> int:  
 return int(self.svm.predict(numpy.array([input]))[1][0][0])  
  
 def predict\_multiple(self, inputs: numpy.ndarray) -> numpy.ndarray:  
 \_, predictions = self.svm.predict(inputs)  
 return predictions.ravel().astype(numpy.int32)  
  
 def save(self, file\_path) -> None:  
 self.svm.save(str(file\_path))

src/eval\_pipeline.py

*# Evaluates the performance of the full pipeline.*import cv2 as cv  
from os import listdir  
from pathlib import Path  
from pipeline import HouseNumberRecognitionPipeline  
from sys import argv  
  
  
IMAGE\_EXTENSIONS = (**'.jpg'**, **'.png'**)  
  
  
if len(argv) != 4:  
 print(**'Usage: python3 eval\_pipeline.py <input\_dir> <recognition\_model\_file> <correct\_numbers\_file>'**)  
 exit(1)  
  
input\_dir = Path(argv[1])  
recognition\_model\_file = argv[2]  
correct\_numbers\_file = Path(argv[3])  
  
pipeline = HouseNumberRecognitionPipeline(recognition\_model\_file)  
  
with open(correct\_numbers\_file) as file:  
 correct\_numbers = eval(file.read())  
  
results = []  
for entry in listdir(input\_dir):  
 file\_path = input\_dir / Path(entry)  
 filename = file\_path.stem  
 if file\_path.suffix not in IMAGE\_EXTENSIONS:  
 continue  
 image = cv.imread(str(file\_path), cv.IMREAD\_GRAYSCALE)  
  
 result = pipeline.process(image)  
 house\_number = **''**.join(map(str, result.house\_number))  
  
 results.append((filename, house\_number, correct\_numbers[filename]))  
  
correct = [r for r in results if r[1] == r[2]]  
incorrect = [r for r in results if r[1] != r[2]]  
  
total = len(correct) + len(incorrect)  
print(**f'Accuracy:** {round(100 \* len(correct) / total, 2)}**%'**)  
  
if correct:  
 print()  
 print(**'Correct:'**)  
 for image\_name, predicted\_number, correct\_number in correct:  
 print(**f'**\t{image\_name}**: c=**{correct\_number}**, p=**{predicted\_number}**'**)  
  
if incorrect:  
 print()  
 print(**'Incorrect:'**)  
 for image\_name, predicted\_number, correct\_number in incorrect:  
 print(**f'**\t{image\_name}**: c=**{correct\_number}**, p=**{predicted\_number}**'**)

src/eval\_recognition\_model.py

*# Evaluates the performance of the digit recognition model.*import numpy  
from pathlib import Path  
from pipeline import DigitRecognitionModel  
from random import randrange  
from recognition\_training\_data import load\_recognition\_training\_data  
from sys import argv  
  
  
CROSS\_VALIDATION\_SETS = 4  
  
  
if len(argv) != 2:  
 print(**'Usage: python3 eval\_recognition\_model.py <dataset\_dir>'**)  
 exit(1)  
  
dataset\_dir = Path(argv[1])  
  
inputs, labels = load\_recognition\_training\_data(dataset\_dir)  
  
random\_seed = randrange(2 \*\* 32)  
numpy.random.default\_rng(random\_seed).shuffle(inputs)  
numpy.random.default\_rng(random\_seed).shuffle(labels)  
  
input\_sets = numpy.array\_split(inputs, CROSS\_VALIDATION\_SETS)  
label\_sets = numpy.array\_split(labels, CROSS\_VALIDATION\_SETS)  
  
for i in range(CROSS\_VALIDATION\_SETS):  
 training\_inputs = numpy.concatenate(numpy.delete(input\_sets, i, 0))  
 training\_labels = numpy.concatenate(numpy.delete(label\_sets, i, 0))  
 test\_inputs = input\_sets[i]  
 test\_labels = label\_sets[i]  
  
 model = DigitRecognitionModel()  
 model.train(training\_inputs, training\_labels)  
  
 print(**f'Round** {i}**'**)  
 for name, inputs, labels in [(**'Training'**, training\_inputs, training\_labels), (**'Test'**, test\_inputs, test\_labels)]:  
 predictions = model.predict\_multiple(inputs)  
 accuracy = numpy.sum(predictions == labels) / inputs.shape[0]  
 print(**f'**\t{name}**:** {round(accuracy \* 100, 2)}**% accuracy'**)  
 print()

src/extract\_house\_numbers.py

import cv2 as cv  
from os import listdir  
from pathlib import Path  
from pipeline import HouseNumberRecognitionPipeline  
from sys import argv  
  
  
NUMBER\_TEXT\_OUTPUT\_FILE = **'House-{}.txt'**NUMBER\_IMAGE\_OUTPUT\_FILE = **'DetectedArea-{}.jpg'**NUMBER\_BOX\_OUTPUT\_FILE = **'BoundingBox-{}.txt'**RAW\_BOXES\_OUTPUT\_FILE = **'RawBoxes-{}.png'**FILTERED\_BOXES\_OUTPUT\_FILE = **'FilteredBoxes-{}.png'**IMAGE\_EXTENSIONS = (**'.jpg'**, **'.png'**)  
  
  
if len(argv) not in (4, 5):  
 print(**'Usage: python3 extract\_house\_numbers.py <input\_dir> <recognition\_model\_file> <output\_dir> [<extra\_output>]'**)  
 exit(1)  
  
input\_dir = Path(argv[1])  
recognition\_model\_file = argv[2]  
output\_dir = Path(argv[3])  
extra\_output = argv[4] == **'T'** if len(argv) == 5 else False  
  
pipeline = HouseNumberRecognitionPipeline(recognition\_model\_file)  
output\_dir.mkdir(parents=True, exist\_ok=True)  
  
for entry in listdir(input\_dir):  
 file\_path = input\_dir / Path(entry)  
 filename = file\_path.stem  
 if file\_path.suffix not in IMAGE\_EXTENSIONS:  
 continue  
 image = cv.imread(str(file\_path), cv.IMREAD\_COLOR)  
 image\_grey = cv.cvtColor(image, cv.COLOR\_BGR2GRAY)  
  
 result = pipeline.process(image\_grey)  
  
 if extra\_output:  
 output = image.copy()  
 for x, y, w, h in result.raw\_regions:  
 output = cv.rectangle(output, (x, y), (x + w, y + h), (0, 255, 0), 1)  
 output\_file = output\_dir / RAW\_BOXES\_OUTPUT\_FILE.format(filename)  
 cv.imwrite(str(output\_file), output)  
  
 output = image.copy()  
 for x, y, w, h in result.filtered\_regions:  
 output = cv.rectangle(output, (x, y), (x + w, y + h), (0, 255, 0), 1)  
 output\_file = output\_dir / FILTERED\_BOXES\_OUTPUT\_FILE.format(filename)  
 cv.imwrite(str(output\_file), output)  
  
 if not result.house\_number:  
 print(**f'No house number detected for** {filename}**'**)  
 continue  
  
 x1 = min(b[0] for b in result.house\_number\_regions)  
 x2 = max(b[0] + b[2] for b in result.house\_number\_regions)  
 y1 = min(b[1] for b in result.house\_number\_regions)  
 y2 = max(b[1] + b[3] for b in result.house\_number\_regions)  
 width = x2 - x1  
 height = y2 - y1  
  
 output\_file = output\_dir / NUMBER\_BOX\_OUTPUT\_FILE.format(filename)  
 with open(output\_file, **'w'**) as file:  
 file.write(**f'**{x1}**,** {y1}**,** {width}**,** {height}\n**'**)  
  
 output\_file = output\_dir / NUMBER\_IMAGE\_OUTPUT\_FILE.format(filename)  
 number\_region = image[y1:y2, x1:x2]  
 cv.imwrite(str(output\_file), number\_region)  
  
 output\_file = output\_dir / NUMBER\_TEXT\_OUTPUT\_FILE.format(filename)  
 house\_number = **''**.join(map(str, result.house\_number))  
 with open(output\_file, **'w'**) as file:  
 file.write(**f'Building** {house\_number}\n**'**)

src/recognition\_training\_data.py

*# Loads the digit recognition training data.*import cv2 as cv  
from pipeline import digit\_descriptor  
import numpy  
from os import listdir  
from pathlib import Path  
from typing import Tuple  
  
  
\_\_all\_\_ = [  
 **'load\_recognition\_training\_data'**]  
  
  
*# Loads the digit recognition training data provided from Blackboard.*def load\_recognition\_training\_data(input\_dir: Path) -> Tuple[numpy.ndarray, numpy.ndarray]:  
 inputs = []  
 labels = []  
 for i in range(10):  
 directory = input\_dir / str(i)  
 for entry in listdir(directory):  
 file\_path = directory / Path(entry)  
 if file\_path.suffix not in IMAGE\_EXTENSIONS:  
 continue  
 image = cv.imread(str(file\_path), cv.IMREAD\_GRAYSCALE)  
 descriptor = digit\_descriptor(image)  
 inputs.append(descriptor)  
 labels.append(i)  
  
 inputs = numpy.array(inputs, numpy.float32)  
 labels = numpy.array(labels, numpy.int32)  
 return inputs, labels  
  
  
IMAGE\_EXTENSIONS = (**'.jpg'**, **'.png'**)

src/train\_recognition\_model.py

*# Trains the SVM used for digit recognition.*from pathlib import Path  
from pipeline import DigitRecognitionModel  
from recognition\_training\_data import load\_recognition\_training\_data  
from sys import argv  
  
  
if len(argv) != 3:  
 print(**'Usage: python3 train\_recognition\_model.py <dataset\_dir> <output\_file>'**)  
 exit(1)  
  
dataset\_dir = Path(argv[1])  
output\_file = Path(argv[2])  
  
inputs, labels = load\_recognition\_training\_data(dataset\_dir)  
  
model = DigitRecognitionModel()  
model.train(inputs, labels)  
output\_file.parent.mkdir(parents=True, exist\_ok=True)  
model.save(output\_file)