Mobile Address Tagging for OpenStreetMap

Joel Nelson David DeBaun Steven Bell

Abstract—We present a solution for tagging features in OpenStreetMap using text recognition with an Android smartphone. The application extracts identifying text from a natural image taken with the phone and uses its GPS to tag the appropriate object on the map. Text detection is performed on the captured image by extracting maximally stable extremal regions, filtering and grouping blobs, and drawing a bounding box around the resulting designated ‘focus’ region. Adaptive binarization is then performed using K-means color selection, HSV thresholding, and connected component filtering. The resulting binary mask is processed using the Tesseract OCR engine, and the text is then displayed for confirmation and subsequently tagged on OpenStreetMap.

I. INTRODUCTION

OpenStreetMap (OSM) is a freely-editable crowdsourced map of the world, which is rapidly gaining in popularity and is used or supported by numerous corporations including Foursquare, Microsoft, Apple and ESRI. While it is possible to draw buildings and streets from aerial photography, a complete map requires that these features be tagged with identifying information such as the street name, house number, or restaurant phone number. This process traditionally requires walking an area, making detailed notes, and then entering tags on a computer. Our application seeks to improve this process by allowing users to simply take a photograph of a building front with their mobile phone to capture the address or other data. By combining text parsed from the image with the location provided by the GPS and existing OSM data, the building can be properly tagged.

Algorithms which extract text from natural scenes can also be used on street-level imagery collections such as Google Street View to radically accelerate the tedious process of tagging buildings with data that otherwise must be obtained from the field. The applications for mobile text extraction in natural images coupled with optical character recognition extend beyond tagging data on web maps, and include sign text language translation, location-based information retrieval, and assistance for the visually impaired.

The primary challenge is to locate text within a complex scene and extract the characters from a background which may include text-like pixels. Once a clean binary mask of the text has been extracted, conversion to a text string can be handled by existing software such as the Tesseract OCR engine.

II. PRIOR WORK

The task of extracting text from natural images has been investigated over much of the last two decades, but it remains a challenging problem. Although humans are very good at locating and reading text, computer algorithms often struggle to find the text, particularly if it contains artistic fonts or handwritten letters. Color variations, shadows, glare, blur, and textured backgrounds all make detection more difficult.

In general, text in natural scenes has strong contrast with the background, and individual letters are areas of relatively little contrast. Characters are usually arranged in horizontal lines, with a consistent color and font across the whole line. Based on these characteristics, recent work on text detection has included one or more of the following approaches:

- Connected-component methods, which operate on a set of potential letter candidates and use heuristics to filter out regions which are not letters. The candidates themselves may be found through either edge or region based algorithms.
  - Edge-based methods take advantage of the contrast at edges to segment letters. Examples include morphological gradients [1] and Canny edge detection [2].
  - Region based methods use both the internal consistency of a letter stroke and its strong contrast to select the pixels which make up the letter. These include maximally stable extremal regions (MSER) and color segmentation [3].
- Learning-based methods, which build statistical models from training data to classify image regions as text or non-text [4], [5], [6]. These methods typically use features based on the texture properties of text derived from convolutional filters or wavelet transforms.

These categories do not constitute disjoint approaches; most successful algorithms use a combination of these.

Our work is based primarily on two previous projects, both of which are region-based connected-component methods. Chen et al. [7] present an algorithm that chooses character candidates in natural images using a combination of MSERs and Canny edges. Non-characters are filtered out based on geometry and the stroke width transform, and remaining candidates are grouped into possible lines of text. Lines of text are parsed into individual words based on classification of inter-character distances as character spacing and word spacing.

Kim et al. use a small ‘focus’ area in the center of the image to narrow the region in which to search for text [3]. Their algorithm uses mean shift color clustering to select three text color candidates in the image, and then thresholds the image with these colors to produce three binary masks, and region filtering is applied to identify an initial existing component. They then perform locally adaptive color thresholding to extract an entire line of text. Character geometry criteria are applied to select the single binary mask corresponding to the true text color.
III. ALGORITHM DESIGN

Our algorithm must be able to locate text anywhere within the captured image and then produce output suitable for the Tesseract OCR engine. Through reading and our own experimentation, we found that Tesseract performs most accurately when provided with a very clean binarized input with characters at least 20 pixels tall.

MSER extraction is computationally expensive, so we operate on a reduced size image. Additionally, our algorithm uses grayscale MSERs, which means that useful color information is ignored. Color extensions to the MSER algorithm exist [8], but the implementation is reportedly much slower than the grayscale version. Together, the result is that the MSERs are not good candidates for feeding to Tesseract.

Conversely, focus-based color segmentation methods perform very clean binarization, but need to know where to begin looking for text. Because it works in HSV space, it is more robust to shadows and lighting variations which cause problems for the MSER algorithm.

A. Text detection

The first stage of extracting text involves a text detection scheme similar to that in [7]. The algorithm flow is shown in Figure 1.

First, Maximally Stable Extremal Regions are extracted from the grayscale image, to exploit the expected contrast in brightness between text and background as well as brightness uniformity within characters. Image blur strongly impacts MSERs, so edge sharpening or Canny edge detection can be used to make the MSER results more robust. Our application gives the user the opportunity to view and retake a photo on the phone’s camera if the image is blurry, so we do not perform any enhancement prior to running MSER.

Next, non-character blobs are filtered out based on their size, aspect ratio, and number of holes. Regions are removed which have:

- Height ≥ 1/2 of the image height
- Aspect ratio greater than 10
- More than 3 holes. The capital letter 'B' and lowercase 'g' (in some fonts) have two holes, and all other letters have one or zero holes. Using a threshold of 3 allows for small defects in the characters, while removing many spurious non-letters. We experimented with morphological closing to remove small defects, but found on our test dataset that this was unnecessary.

The candidate characters are then grouped into lines, using the proximity of their centroids relative to their width. Character candidates which have a height ratio of more than 2 are not joined, since normal text has relatively consistent font size, even taking capital and lowercase letters into account.

Once characters have been grouped, the groups are filtered based on the properties of text lines. Groups containing only a single character candidate are discarded. We fit a line to the centroids of remaining groups, and calculate the mean-squared error (MSE). Groups which do not form a line have high MSEs and are removed.

In many cases, fences or window panes can produce a row of boxes which appear similar to a line of text. To remove these, we use the observation that most text has low solidity, that is, the ratio of a character’s pixels to the area of its bounding box is typically less than 0.5. Letters such as lowercase ‘l’ have solidity approaching 1, but the mean across an entire line of text will be lower. This assumption fails on very bold fonts or unusual words (e.g., ‘ill’), but it works well on our test dataset. An example of this is shown in Figure 2.

The groups which remain after this filtering are assumed to contain text, and a bounding box is drawn around them.

B. Adaptive binarization

The second component of the text extraction algorithm, outlined in Figure 3 uses the technique from [3] to perform locally adaptive binarization of ‘focus’ window connected components. The bounding box region obtained from the prior text detection step becomes the ‘focus’ area for performing K-means color clustering on the original color image in RGB space. K-means clustering is used instead of the mean shift clustering used by Kim et al. [2] due to the availability of existing K-means functions in Matlab and OpenCV. This produces three primary seed colors, which are then used to perform color thresholding on the ‘focus’ region. Similarity of seed colors in the HSV space is measured using HSV distance.
as developed in [3]:

\[
D_{HCL} = \sqrt{A_L (l - l_s)^2 + A_{CH} (c^2 + c_s^2 - 2c_c\cos(h - h_s))},
\]

where \( A_L = 0.1, A_{CH} = 0.2 + (h - h_s)/2 \), \( A_L \) is constant of linearization for luminance, and \( A_{CH} \) is a parameter for reducing the distance between colors of the same seed color hue [2].

In order to determine which of the seed colors and its corresponding mask belong to the text, robust locally adaptive color thresholding is performed. First, binarization is performed by keeping pixels that have an HSV distance of less 0.08 from the given seed color. The binary image is then dilated using a 5x5 rectangular structuring element to remove small holes. Non-character components are filtered out by imposing aspect ratio and compactness criteria for each mask, and an initial existing component is identified.

The search is then expanded to find neighboring components and an adaptive binarization is performed using a sliding window approach. The window is defined to have 1.5x width of the initial component and the height of the focus region. Each time a component is found, the seed color is updated to be the average color of the newly found component. This allows the binarization to be more robust against small color variations across the image. Searching stops when no more components are found that satisfy aspect ratio and compactness constraints.

The final text decision is made between the three resulting binary masks by checking number of components in the mask, their height, compactness, and distance of separation between components. The following criteria are used to determine if the mask contains text:

- Number of components greater than 2
- Variation of heights of components less than 0.5
- Distance between components less than half the width of initial component

The correctly selected binary mask of the text is then passed to Tesseract and the characters themselves are parsed. Figure 4 shows the steps in the binarization process.

**IV. IMPLEMENTATION**

A. Matlab implementation

We initially developed both algorithms described above in MATLAB. We also implemented the stroke width transform [2], [7] and explored it as a criteria for grouping objects. Because we do not use an edge detection method to create clean lines on our letter candidates, letters occasionally joined together or created nearly closed blobs, causing the SWT result to be unreliable.

B. OpenCV implementation

In order to run our algorithms on a phone, we ported them to OpenCV. The adaptive binarization algorithm was fully implemented on the Android platform using the OpenCV 2.4.0 library. We used the Java OpenCV bindings, which enabled the entire application to be written in Java.

Unfortunately, the OpenCV MSER function is not fully accessible from Java, so we instead ran the text detection code in Python on a PC. We also had to make a small change to the OpenCV library in order to select between light and dark MSERs in our code.

We used a simple PHP script running on a web server to receive an uploaded image and pass it to the Python code. The results are passed back from Python script to the client.

C. Android application

Our application consists of four panes: a map view where the user can select a building, a viewfinder for taking a picture, a progress screen displayed while the image processing code runs, and a final results page. For simplicity, we use the stock Android camera application as the photo-capturing pane. Example screenshots of our application are shown in Appendix B.

To display untagged buildings in a useful and natural way, we created a custom basemap using OpenStreetMap data where buildings without street addresses are highlighted. The map implementation is described more fully in Appendix C.

**V. RESULTS**

To test the text detection algorithm, we ran it on a set of twelve images from a test dataset captured with the phone. In the majority of the images, no false positives are detected. The overall performance is 18/24 text lines detected (75%) with an average of 1.25 false positives.
VI. CONCLUSIONS AND FUTURE WORK

The system in its current form has several shortcomings. First, we did not get to the point of being able to run both phases of the algorithm together on the phone, and it would be better to consolidate all of the processing on the server or on the phone. The run time of the adaptive binarization algorithm on the Android phone is relatively slow. Using a 230 x 1990 ‘focus’ region, the program completed in approximately one minute. The greatest slow down occurs in the JNI calls from Java to retrieve each pixel value in the image matrix. Since thresholding is performed on three images, the bulk of the processing time is spent calculating HSV distance between pixels.

We were unable to evaluate the performance of the MSER detection algorithm on the phone, but it runs in 0.6 seconds on a 2.0 GHz Intel Core 2 laptop, as measured with the Unix time utility. The solution would be to code both algorithms in C++, where all OpenCV functionality is exposed and pixel accesses are inexpensive.

REFERENCES


APPENDIX

A. Work breakdown

Dave developed code to take pictures, read the phone’s GPS and orientation sensors, and captured the test dataset. He also experimented with Tesseract to determine its robustness. Joel developed the adaptive binarization algorithm in MATLAB and on the Android phone. Steven developed the MSER-based text detection algorithm in MATLAB and Python. He also created the OSM map tiles and the framework for the Android application. We all worked collectively on the poster and report.

B. Application screenshots

Example screenshots of our application are shown in Figure 5.

C. Map design

To create the basemap, we imported the OpenStreetMap data for California into a PostGIS database, and created a custom map style in TileMill [9]. Buildings with the OSM "addr:housenumber" tag are colored gray, buildings without are colored light red. Tiles for the Stanford area were rendered with Mapnik and stored on Stanford AFS webspace. The Android client application has an embedded WebView, which runs a JavaScript application built on the Leaflet API [10] to display the map. Sample map tiles are shown in Figure 6.
Fig. 5. Screenshots demonstrating the workflow of our application. The user is first presented with a map showing their present location. When the user double-taps on a building, the camera app opens and the user takes a picture. The third screen shows a progress bar as the application waits for the image processing to complete. In the final screen, the user can view and confirm the parsed text.