UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

ZAKLJUČNA NALOGA
(FINAL PROJECT PAPER)

SEGMENTACIJA KANE IN KITAJSKIH ZNAKOV IZ KOMPLEKSNIH PRIZOROV (SEGMENTATION OF KANA AND CHINESE CHARACTERS FROM COMPLEX SCENES)

JANI JAMES SLAWECHKY

2020
ZAKLJUČNA NALOGA (FINAL PROJECT PAPER)
SLAWECHKY

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Segmentacija Kane in Kitajskih znakov iz kompleksnih prizorov

(Segmentation of Kana and Chinese characters from complex scenes)

Ime in priimek: Jani James Slawechky
Študijski program: Računalništvo in informatika
Mentor: doc. dr. Peter Rogelj
Somentor: asist. Jordan Aiko Deja

Koper, avgust 2020

Ključna dokumentacijska informacija

Ime in PRIIMEK: Jani James SLAWECHKY

Naslov zaključne naloge: Segmentacija Kane in Kitajskih znakov iz kompleksnih scen

Kraj: Koper

Leto: 2020

Število listov: 30 Število slik: 14 Število tabel: 1

Število referenc: 13

Mentor: doc. dr. Peter Rogelj

Somentor: asist. Jordan Aiko Deja

Ključne besede: OCR, Japonščina, Kitajščina, tekstovna reprezentacija, segmentacija, tekstovno restrukturiranje

Izvleček:
Ker OCR sistemi vsebino pogosto napačno klasificirajo, kjub predhodnimi predelavami, je zato zaželeno imeti čim večji nadzor nad njeno reprezentacijo za pridobitev najboljših rezultatov. Ta projekt bo torej predstavil korake za segmentacijo in pridobitev individualnih znakov iz kompleksnih prizorov, kateri nam bodo nato omogočili restrukturiranje vsebine v katerokoli obliko. Te se bodo lahko tudi uporabile pri dodatnih transkripcijah ali morebitnih dodelavah, ki bi služile za izboljšavo rezultatov od izbranega OCR sistema. Večina vsebine se bo nanašala na Japonščino, vendar zaradi njene kompozicije, se ideje lahko aplicirajo tudi na druge strukturi podobne jezike kot so Kitajščina ter Vietnamščina (izključujoč dodatnih Japonščini specifičnih obdelavah).

Key document information

Name and SURNAME: Jani James SLAWECHKY

Title of final project paper: Segmentation of Kana and Chinese characters from complex scenes

Place: Koper

Year: 2020

Number of pages: 30 Number of figures: 14 Number of tables: 1

Number of references: 13

Mentor: Assist. Prof. Peter Rogelj, PhD

Co-Mentor: Assist. Jordan Aiko Deja

Keywords: OCR, Japanese, Chinese, text representations, segmentation, text restructuring

Abstract:
Optical Character Recognition systems are prone to miss-classifications, even when presented with clean preprocessed input. Therefore it is in our best interest to have as much control over the data we provide them, so as to obtain the best possible results. Therefore this project will propose a series of operations for segmenting and obtaining a set of individual characters from a scene. Which could then be rearranged as required, and used for any further transcriptions or additional post processing steps meant to increase the chances for a correct classification from our OCR system of choice.
Although the majority of this project will deal with Japanese text, given its structure, the same can be applied to other languages written in simplified and traditional Chinese, like Mandarin, Cantonese or Vietnamese (excluding some language specific post processing).

Acknowledgement

I sincerely thank, all the people who took their time, to write all the in-depth examples and documentation for the tools I discovered while tackling this project. As well as my mentor and co-mentor who agreed to be a part of this.

List of Contents

List of Tables

List of Figures

List of Abbreviations

e.g.	for example
OCR	Optical Character Recognition
CEG	Character Element Grouping
CNN	Convolutional Neural Networks
RNN	Recurrent Neural Networks
LSTM	Long Short Term Memory Model
NLP	Natural Language Processing
RGB	Red, Green, Blue Color Model
RGBA	RGB with Alpha channel
IC spacing	Inter-Character spacing

1 Introduction

The main motivation for this system, is the lack of high precision optical character recognition (OCR) engines that can be used on complex images. Images that are not solely comprised of text. Although the prime target are color images, transcribing greyscale images, such as comic book pages, might also be desirable.

The languages being targeted in this project are notoriously difficult when it comes to OCR, as they are comprised of an exceptionally vast number of symbols (graphemes called logograms), especially when compared to alphabetic systems. The major open-source OCR engines are based on Recurrent Neural Networks (RNNs), such as Long Short Term Memory (LSTM) [8], for tackling them, which can also be seen commonly used with other Natural Language Processing (NLP) problems, such as Machine translation. This type of approach, can and will occasionally produce incomprehensible outputs, where the number of transcribed glyphs might even exceed the physical constraints of the input image, providing more characters that could actually fit by injecting multiple in places where there are none.

In response to this, the described method is designed to construct character representations beforehand, before giving the data to the engine, allowing us to perform sanity checks, such as basic symbol count comparisons between the expected number of characters in the input data and the number present in the transcription. These representations, which would include additional information on character composition, keeping track of all segments in a glyph, would thus also give us more options for correcting any per character miss-classifications, by providing us with the ability to use other tools in conjunction to OCR on specific characters, for better final results.

And lastly, as text in complex images can have uneven indentations, be in vertical-flow or be center aligned, all of which presents difficulties when it comes to optical multi-line recognition, the existence of such a text abstraction is essential to restructure it into one of the formats that OCR engines are known to work best with. Such as greyscale single line examples.

1.1 Language specifics

The language on which we will mostly focus, will be Japanese, which employs multiple writing systems; most of which are fully or in part component based. Meaning that, a substantial amount of individual graphemes can be split into multiple segments (for logograms referred to as radicals), some of which may be valid glyphs on their own. Although the latter only applies to kanji (a variant of traditional Chinese characters), kana (the language's syllabary writing system) has also its share of other unique quirks.

In conjunction to this, the language also supports multiple valid text-flows / directions:

1. horizontal : characters written from left to right; lines written from top to bottom
2. vertical : characters written from top to bottom; lines written from right to left

Some graphemes are dependant on text-flow, appearing in different orientations and positions in regards to it. Most notably 長音・ー; the vowel extension sign, and standard punctuations 、。「」.

Most typed text present in complex scenery uses rectangular shaped characters, with ratios $< height/width >$ somewhere in the range of (0.9,1.4), while also keeping consistent spacing rules. Both choices that can occasionally be broken, with the inclusion of half-height and half-width glyphs representing kana characters (や→ﾔ カ→ｶ).

Additionally, Japanese, as well as the others, is also unique in the fact that it doesn't hyphenate words that are split at line endings, making it a requirement to always check adjacent lines if the glyph we are inspecting is at an edge.

The notion of a descender is also absent in the writing system, so knowing the baseline of one character is sufficient for extrapolating the baseline of the whole line, as long as that symbol is a Chinese logogram (kanji / hanzi) and not one of the previously orientation dependent ones, a specific numeral like 一、 二 (depending on typeface) or an iteration mark such as 々.

It should also be noted, that the writing system in general, does not use spaces as word delimiters, making word splitting somewhat more complex, but can still be seen having them present, for stylistic reasons.

1.2 Complex Image Characteristics

Before continuing, there are a set of observations that can be made when it comes to complex scenes / images, which will in turn be used for reasoning regarding each step of foreground segmentation and character creation. Those being:

A For visibility reasons the foreground and background colors should be distinct : this is often enforced by a text border (Figure 1)

B The text might not be of a single color, it might have a gradient going from top to bottom or side to side (Figure 1)

C Images very rarely have areas of pixels with exactly the same color

D Continuous text has no more that a character size of line spacing (Figure 2)

Figure 1: Image depicting characters with color gradients and border

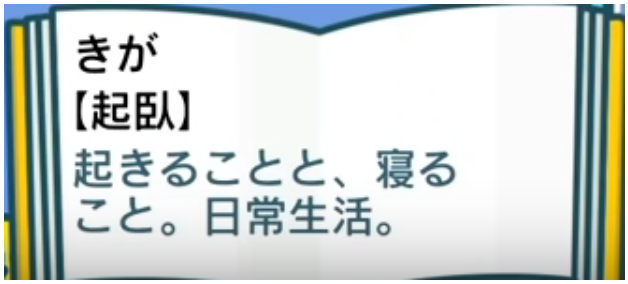

Figure 2: Image depicting multi-line text with much less that a character of line spacing

2 Segmentation Method

To produce the representations mentioned in chapter 1, from now on being referred to as Character Element Groupings (CEGs), we will perform the following steps:

1. Image quantization
2. Character Discovery
3. Text-flow deduction
4. Line partitioning
5. Multi-line detection and partitioning (optional)

2.1 Image Quantization

According to point A listed in the observations (Chapter 1.2), regarding the distinction of text and background colors, we can assume that there might exist a clustering, such that in some color-space, will keep the foreground (text) colors, irregardless of the notion presented in point B, dealing with possible gradients, under the same label while giving the immediate border a distinctly different label.

Granted that this is correct, we can attempt to divide the whole image into k color groups using a vector quantization method such as **k-means clustering** [6], so that there exist at least two groups, one containing our foreground and a different one with the immediate background.

But k-means also presents a problem, as it requires knowledge of the number of clusters needed for a sufficient separation beforehand. We can attempt to quantize the whole image into e.g. 16 groups, but that might either result in too many, where we split any gradients or other color variations (point C) present in the text into separate labels, or too few, requiring additional clustering.

So to completely bypass this, we can attempt to create a binary quantization tree (Figure 3) of the image at hand, in a depth first manner, by segmenting each node into two groups of distinct but internally similar colors. From here we can continue this process, by performing it recursively on each child node, only taking into consideration the subgroup of colors it has. Doing so eliminating any possibility of over segmentation

while searching for possible text, as at no point will we ever reach a node, having both the border separated while also splitting the foreground, as that would imply that one of the foreground colors was closer, in terms of it's L^2 (Euclidean) distance, to its immediate background than the rest of the text.

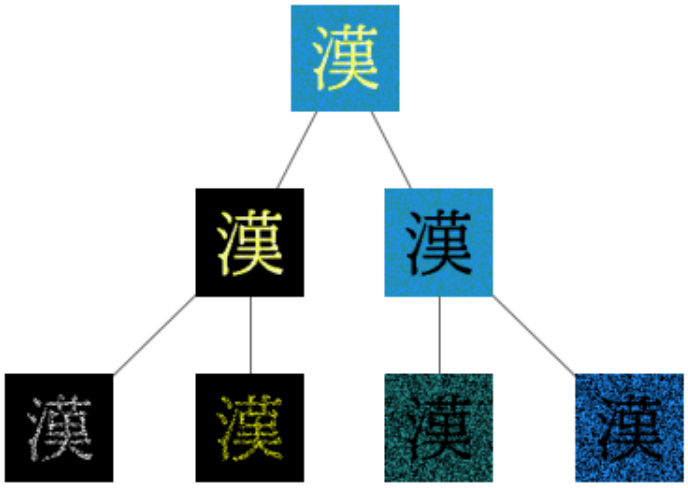

Figure 3: A visual representation of a full binary quantization tree. The top being the full image with all four colors mixed together, with the second level having grouped the closest while excluding the rest (*black*) which can be found in its neighboring node. The final being all 4 colors individually separated.

Having performed at least one round of binary quantization, thus splitting the previous node into two and selecting one of them, we can continue to the next stage.

2.2 Character Discovery

As the previous clustering step does not guarantee the sole presence of text, but may also contain any other background segments that had a similar color palette, we will need to perform a set of specific operation for extracting only the needed data. As long as no glyphs (representing individual graphemes) are connected, we can perform contour detection for obtaining all regions that will be used as elements in performing character discovery.

Character discovery being a process of joining these elements (components) together in such a fashion, that we end up with a selection (a set) that most closely fits all the following criteria:

1. The produced set has to be within the previously specified character ratios
2. The set must not be the only character present in the image (usually resulting from either being too large or having no neighboring glyphs)
3. The set's neighboring elements must fit character / outer spacing constraints (all of its immediate neighbours, that are part of other valid glyphs on the same line, should be no farther than 2 characters away; or in other words, a maximum of two spaces between glyphs, the way text in novels is usually presented)
4. The set's inter-character spacing must be in accordance to its neighboring elements (inter-character is expected to be smaller than the spacing between adjacent characters)

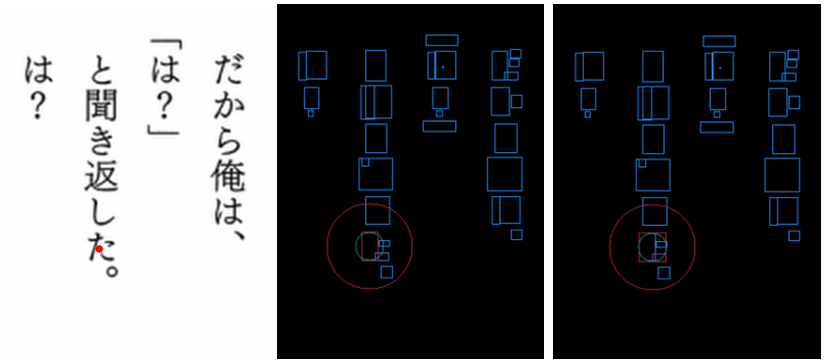

Figure 4: A successful construction of a set fulfilling all the needed criteria. With a seed-point in *red* (an optimization technique explained in section 2.5.3), for further explanations regarding symbols and colors see Figure 7.

2.2.1 Set construction caveats

Although very rarely, character glyphs can have inter-character (IC) spacing that exceeds that of outer (as illustrated in Figure 5). To limit problems that might result from this, an initial check is performed for full character elements present nearby. Those being, single elements representing the full graphemes, as they have the desired ratio from the start and have no notion of IC spacing, as its by definition not applicable (see Figure 5, the square box on the right containing the character 人, as well as the character 働 as represented in Figures 6 and 7).

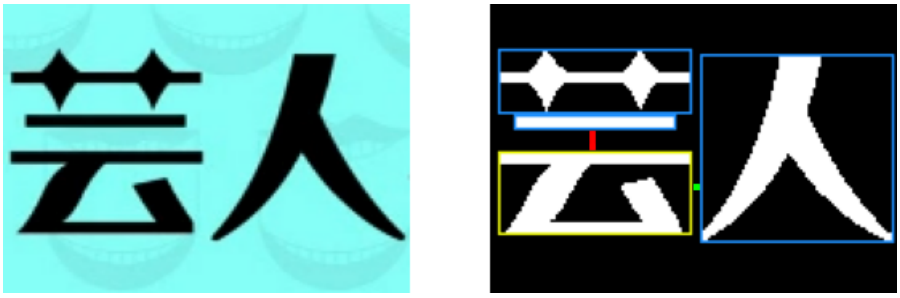

Figure 5: Character glyph displaying smaller spacing than is expected; $Yellow \rightarrow$ initially selected element, $Green \rightarrow$ character/outer spacing and $Red \rightarrow$ IC spacing

The same procedure may also be applied, when an already selected set cannot fulfill any of the other previously stated criteria (for example, when it comes to ratio see Figure 7), so that the selection might be moved to them instead.

When all options are exhausted, and if no presumed text is detected, we can either continue by switching to a different branch of the tree or split the current node into two, repeating the whole process anew.

2.3 Text-flow deduction

After successfully obtaining a character set, text line construction can begin. This is a process of aggregating elements in both the horizontal and vertical direction into their respective groups, terminating when spacing is exceeded. These aggregations are then inspected and their outer spacing calculated.

By comparing both vertical and horizontal, we can select the aggregation with the smaller distance between its elements, as the one dictating text-flow.

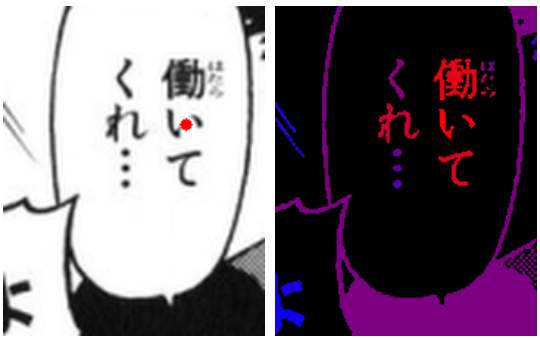

Figure 6: Left being a greyscale image with vertical text-flow, containing a seed-point in *red*. While on the right, an image representation of contour distances from said seed-point, where the gradient is closest to farthest as $red \rightarrow blue$.

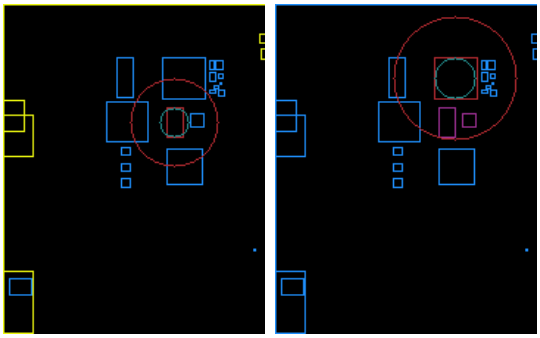

Figure 7: The process of failed character discovery, which was started from the *red* rectangle on the left, resulting in using a different element in its proximity as seen on the right (described in section 2.2.1). *Cyan* circles indicate the minimum character boundary of the selected element set. This is the area that should only be occupied by the character itself. The *red* circles are meant to visualize a single character of spacing from our selected set, where the first neighboring elements should intersect. *Magenta* is used to indicate already tried elements that didn't fulfil the needed criteria, either individually or as a set, before moving to a better position.

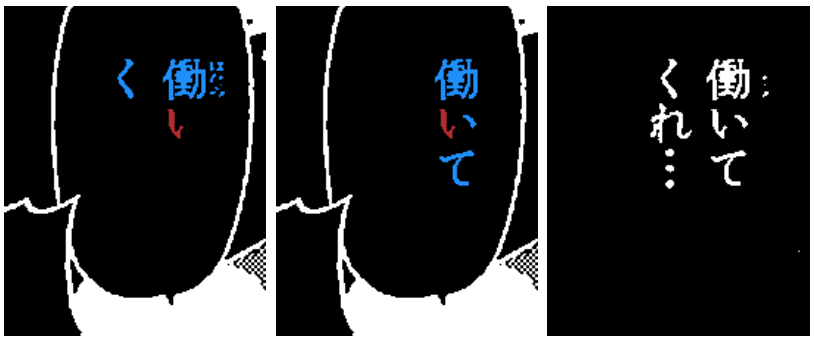

Figure 8: The first two images are depicting text-flow deduction in both horizontal (on the left) and vertical directions (in the middle), with the final segmented image (right), with all lines of text exposed in black and white.
The *blue* elements are the previously described aggregations pertaining to each direction, based on the selected starting element set, which in this case is the character above the *red* segment as decided by in Figure 7.

2.3.1 Multi-line detection

Although not strictly necessary, we can already perform multi-line detection and extraction. As sequential lines of text relating to the same content are almost always of the same font, having equal proportions / ratios, we can easily find them by scanning in parallel to our selected line of elements, keeping in mind character spacing constraints and Point D from section 1.2 referring to line spacing.

2.4 Line partitioning

Having the foreground now fully separated, with at least one line of text exposed, we can begin to split this into our final CEGs line by line. This is achieved by analyzing empty areas between the elements of a particular line and using them as indicators for correctly joining segments based on our initial characters dimensions. As long as all characters are of equal proportions, we are guaranteed a correct final arrangement of all contours into their respective element groups.

The contours present in these final groupings can now be redrawn as required, keeping all inter-character element spacings intact.

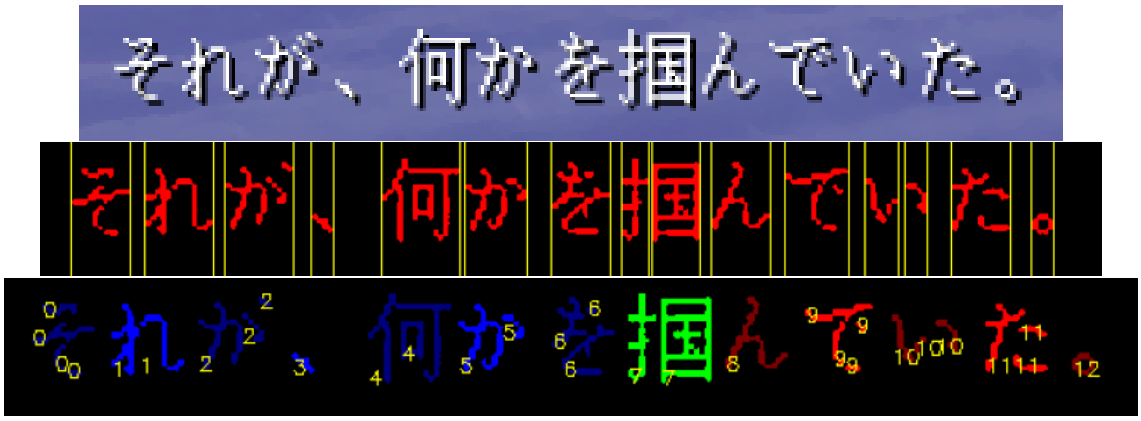

Figure 9: The partitioning operation visualized, with the final CEG contours numbered and colored depending on position. *Green* being the selected set (or more precisely the set containing the provided seed-point).

2.5 Implementation specifics

As all the previously described steps have collectively drastic time complexity, using the mentioned trial and error approach for character discovery, with the only false-positive deterrent being the actual OCR engine at the end. Besides this, as most use-cases in need of a system such as this, would not require full text transcriptions, but quick specific character results, we will tackle the implementation in regards to a user provided point on the image, which would correspond to a word transcription that should be provided back.

This would in series both guarantee presence of text in the image, while also specifying a point that a glyph's bounding box should encompass.

2.5.1 Image types and quality

Images are almost exclusively encoded as 3 dimensional arrays of $h * w * e$, where e is a vector of length 3 or 4 (RGB[1] and RGBA[2] respectively) with a common depth of 8 bits per element (known as 8bit color - 256 values per channel). In cases where the initial image is already in RGBA format, we remove the alpha channel as it can be used elsewhere in our segmentation process (later described in section 2.5.2).

There is also no guarantee, that the image as provided to the system, will in fact be of a sufficient quality, so as to not contain artifacts which might be represented as minor pixel color deviations, that could interfere with the continuity of text borders. This might consequentially cause pathways connecting the foreground text to its immediate similar surroundings (as depicted by white area in the middle two images of Figure 10).

To mitigate this, I propose resizing the image 2 fold via Lanczos interpolation (OpenCV LANCZOS4 [9]) in the hopes of overshadowing such *pixel deviations* while also giving ourselves some leeway for performing prior erosion before discovery, with the same goal of eliminating these connections. Since erosion is a destructive operation that can eliminate finer details present in the text (see Figure 11), we should perform this as a separate segmentation, ran in parallel.

[1]Red Green Blue Color Model
[2]RGB with Alpha Channel

Figure 10: Original image → no modifications → rescaling → rescaling and eroding. The *white* area in Figure 10 denotes background and implies connectivity, whilst *red* signifies the currently selected glyph or part thereof. The color *green* being elements outside the horizontal aggregation as a consequence of not fulfilling spacing requirements and *blue* hinting other valid ones that could be representing characters. Performing no additional operations completely leaves out the primary symbol in question (*red*), with only the final example, running all of the aforementioned steps, returning the full expected result.

Figure 11: Depicting loss of detail in final segmentation by using erosion: Original image → rescaling → rescaling and eroding

In cases where the text borders are not sufficiently pronounced, it is also recommended to perform basic prior sharpening for creating higher contrasts in such areas, exposing previously unclear borders. This problem was most commonly encountered when segmenting images from older digital media. Greyscale images, should also be exposed to the same prior sharpening, as that equally seems to produce better results.

The final system should perform at least 4 types of discovery : 1. using no erosion or sharpening, 2./3. using only one of each, 4. using sharpening and at least a single iteration of erosion. All types should be ran in parallel with only the option providing the highest confidence levels be returned back to the user.

2.5.2 Quantization caveats

After performing the initial clustering, every sequential one will have areas of no valid color. These areas should all be labeled accordingly, either by setting them to a specific value, so that it does not interfere with the other two (see Figure 3 color *black*), or by setting the opacity of color areas to its maximum and making all removed area transparent. Although extremely unlikely, this can cause a problem when clustering pure black text after the initial round, as that might get labeled as invalid / removed area depending on the rest of the image (the minimum L^2 distance being 255 between pure black [0,0,0,255] and transparent [0,0,0,0]).

2.5.3 Character Discovery based on seed

Provided we have some initial point on the image, corresponding to a specific character, we can perform discovery in a similar manner as described in section 2.2. This being done by selection the closest contour / element based on the seed point (using OpenCV pointPolygonTest), with the caveat being, that we are now constrained to it and can't easily move to a more advantageous position as would be otherwise possible. Moving to other elements can of course still be performed, but might needlessly require additional line detection if we mistakenly select a character from an adjacent line or some other image segment not related to text. Multi-line detection should only be necessary when it comes to single word transcriptions, when our selected set is located at the line's start or end, indicating that it could actually be a part of a word that is spanning multiple lines, requiring line concatenation.

2.5.4 Extended Line partitioning

To save on complexity, we can implement both horizontal and vertical partitioning as the same operation, by just rotating the image representing vertical text-flow 90°, so it resembles horizontal. This will of course leave us with all the characters being in their incorrect orientation, which will have to be corrected after partitioning. Recalling the details from section 1.1, we now have to take into account text flow dependant graphemes, as some will already be correctly positioned and oriented. Although this set is already relatively small, we might wish to lower it further down by excluding punctuations all together, as they have little value when it comes to word detection. Leaving us only with the vowel extension sign. The specifics of which are quite simple, as it has a unique ratio not shared with any another character except the similar looking numeral one 一 which doesn't change orientation depending on flow.

In most cases, regardless of prior efficiencies, we will be confronted with background remnants, presenting difficulties finding empty spaces between symbols. To deal with this, where a spacing is expected, given the previous elements, we can try to find it using a path-finding algorithm such as A* [10]. Whose path information can then later be leveraged while discerning the actual importance of the elements being circumvented, in relation to CEG creation. And finally, provided that all our partitioning heuristics fail, the path can also be used for guidance when user intervention is required (e.g. providing options for contour removal via some interface).

2.5.5 Shortcomings

The hardest step in the whole segmentation being discovery, there are many occasions where the system might incorrectly deduce its initial character sizing, leading to further inaccurate decisions. Examples being mostly related to kana selection, as it is known to be used with different proportions and positioning, compared to the rest of the Chinese (logographic) characters that might be present (such as seen in Figure 9).

3 Post-processing CEG applications

This section will demonstrate additional benefits of possessing such a text representation (as already hinted in chapter 1), by presenting a simple example of CEG usage, using Tesseract [8] as the OCR engine, MeCab [1] in conjunction with the UniDic [2] dictionary for morphological analyses, such as splitting the text into individual words, and a collection of Japanese dictionaries (JMDict [4], Shinmeikai, Daijirin [5]etc.) as well as frequency lists for word validation.

To start, we will construct a set of Convolutional Neural Networks (CNNs) [13] meant for traditional character dictionary lookup, for use on each language's logograms (in this case only Japanese). The most common approaches being 4-corner [3], SKIP and stroke count. We will focus on the 4-corner method, as it requires the least training time and input resources for achieving a model of sufficient accuracy. The training input for which is to be generated from commonly available webfonts, while the final model be constructed and trained with the help of Tensorflow [11] and the Keras API [12].

The core pipeline being an follows:

1. Perform segmentation based on a seed-point
2. Obtain initial CEGs of the selected line (or multiple depending on position)
3. Construct an input image as a single line using the CEGs
4. Perform multiple types of single-line transcription (using Tesseract[1])

Having performed the previous steps, as an example, let us presume the following transcriptions. One with the values $A^xC^xA^xD^xB^xA^xE^xD^x$ (with A, B, C, D, E being unique characters from string x) and another as $B^yA^yC^yD^yB^yA^yF^y$. By performing sequence matching and re-transcribing restructured input files, we might end up with the final representation like $?AC?DBA(E|F)?$ (with ? denoting symbols that both methods were not able to successfully classify). Now with the help of dictionaries and frequency lists, we might be able to conclude that the output could either be $?ACCDBAE?$ or

[1]As of version 4.1.1, those being PSM : 7 = Treat the image as a single text line, 13 = Raw line. Treat the image as a single text line, bypassing hacks that are Tesseract-specific.

?$ACDDBAF$?. To finally decide on an option, we use the previously mentioned CNNs to predict all corners of both the 4th($C|D$) and 8th($E|F$) symbol, resulting in a final decision being ?$ACDDBAF$?. To finish this off, we can morphologically parse and split the data into separate words e.g. AC and $DDBAF$, which if both valid, can then be annotated on the image.

The initial performance of the CNN models (as described) is somewhat expected to be lacking, as the training data might have little resemblance when compared to the real world extractions. Therefore it is recommended (if such technique is to be used) to be constantly performing additional online training, by using correctly transcribed symbols as well as any user corrections.

In cases where usage of such models is not applicable or desirable, different types and amounts of restructuring, such as changes in padding and character spacing, while systematically providing only subsections of the full extraction, although slower, can still be known to produce similarly good results.

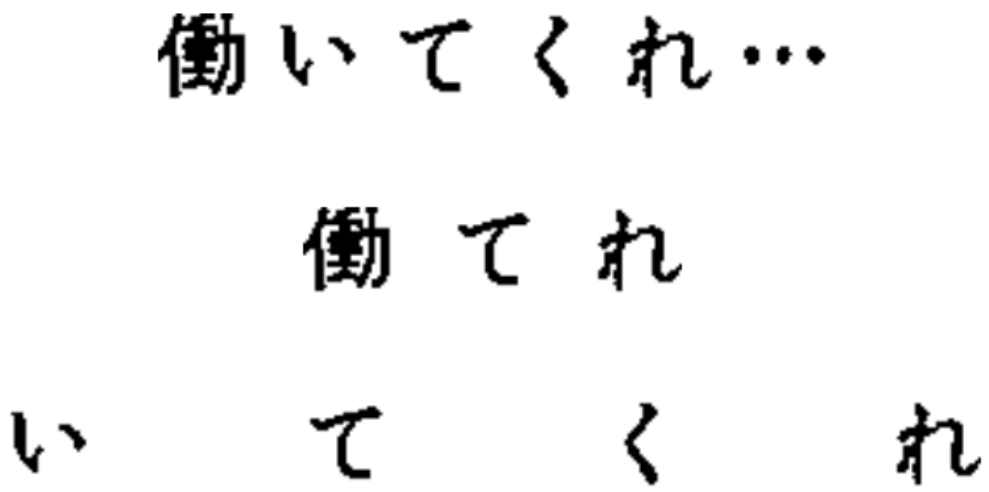

Figure 12: Examples of CEG generated input files: 1. being the standard normal output of the text from Figure 6, restructuring vertical text-flow to horizontal, concatenating lines, with an optionally rotated ⋮, 2. arbitrary restructuring with only even indexed characters present (starting from 0), 3. listing all characters from the selected one onward, excluding punctuations, with a custom enlarged outer spacing

4 Results

Approach	**Result**
Tesseract only (text)	C 22.34% G 6.24%
Tesseract + Segmentation (text)	C 45.69% G 5.71%
Tesseract + Segmentation (text) (CEG constructed line)	C 86.21% G 63.11%
Tesseract + Segmentation (CEG) + MeCab + 4-corner CNNs (text)	C 87.38% G 65.41%

Table 1: Final results (tested on the image dataset described in section 4.1). *C denoting Color images, while G refers to Greyscale*

The final implementation used for evaluation was created using the OpenCV library [7]

The testing data is constructed from manually inspected segments of an image, created by the segmentation pipeline, containing all the steps till partitioning. This describes the area of the selected text as a type of bounding box which is then presented to all 4 methods, evaluating their transcriptions.

The final results are based on both the confidence levels as well as the amount of text that was transcribed. [13]

4.1 Dataset

The full dataset is comprised of 87 images spanning a number of different mediums, from movie stills to video game screenshots, pages from novels and comics. [13] The majority of the set are colored images (approximately 3/4) with mostly horizontal text-flow, while the greyscale images are in most part from comics and novels and have therefore vertical flow.

Figure 13: Examples of typical vertical greyscale images present in the dataset

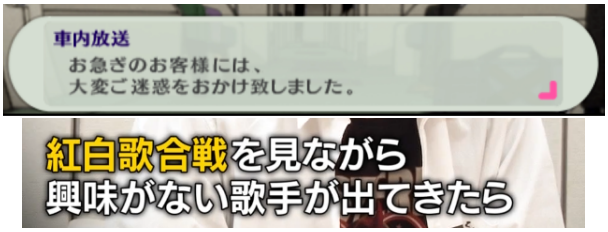

Figure 14: Examples of typical horizontal colored images present in the dataset

5 Conclusion

In summary, this project was meant to present the value of extracting and restructuring text from complex scenery, so that we can obtain better results when it comes to OCR. Although the benefit of this might be clear now, it bares noting, that the actual implementation of a fully automatic system, would require employing some sophisticated heuristics when it comes to character discovery, which were out of the scope of this paper. The system as mentioned still heavily relies on OCR for discerning if it is on the right path of the tree in regards to where text is. This can still lead to incorrect transcriptions and long running times, so is therefore best used with minor user intervention helping it along the way. An idea for how to improve this, would be to construct and attempt to train a character detection model which would be used on each node of the tree in the hopes of obtaining more insight before attempting character discovery.

6 Povzetek naloge v slovenskem jeziku

OCR sistemi so nagnjeni k napakam, še posebej značilno pri transkripcijah azijskih jezikov. Med katerimi so naj izraziti tisti, ki vsebujejo Kitajske znake ali njihove variacije. Saj ti imajo še posebej ogrmono število unikatnih simbolov, v primeru z sistemi baziranimi na abecedah. Vendar cilj tega projekta ni le izboljšava standarnih transcripcij, vendar način uporabe OCR na kompleksnih prizorih, kateri ne vsebujejo le besedila, ter ne glede na to če je celotna vsebina enobarvna ali več. Projekt je posebej orientiran na en izbed jezikov, Japonščino. Vendar zaradi njene sestave, saj prav tako vsebuje Kitajske znake poleg svojih, se lahko podane ideje aplicirajo tudi na druge strukturi podobne, kot so jeziki, ki uporabljajo samo enostavne ali tradicijonalne Kitajske simbole. Omenijo se tudi glavni aspekti pogosto uporabljenih pisav in tipov črk, ter generične značilnoski kot so večdelni logogrami ter prisotnost različnih smeri pisav.

Projekt v celoti predstavja postopek segmentacije in pridobitve individualnih znakov iz prizorov, katere lahko preredimo v katerikoli format vsebine, ki najbolj usteza določenemu OCR sistemu. V tem projektu je to preoblikovanje v eno samo vrstico, saj ta predstavlja najboljše rezultate ko pride do uporabe Tesseract sistema.

Temeljna ideja sloni na segmentaciji prizorov glede na podobnost barv, kjer je predpostavljeno glede na opazovanja, da tekst in vsebina ki ga obdaja sta barvno različna. To je včasih zagotovljeno tudi prek uporabe obrobe. To segmentacijo opravljamo prek posebnega pristopa, kjer razdeljujemo prizor rekurzivno v dve barvni skupini z *k-means* metodo, ter s tem postopkom kreiramo kvantizacijsko binarno drevo z vozlišči, ki vsebujejo le podobne barve. To tudi prepreči prekomerno segmentacijo, saj vsebina določenega vozlišča, ki vsebuje besedilo, se nikoli ne bo porazdelila v dve, tako da bi obenem odstranila obrobo ter hkrati del teksta. To bi seveda pomenilo da je del barv pisave bil bližji obrobi / ozadju kot pa preostanku osprednega teksta. Po izbiri začetnega vozlišča v drevesu pričnemo z iskanjem in združevanjem, povezanih elementov ki so prisotni. Ta postopek poskuša sestaviti nekaj podobnega znakom, glede na vnaprej določene kriterije. Če je ta korak neuspešen, prestopimo v sosednje vozlišče ali pa nadaljujemo s segmentacijo trenutnega ter ponovimo postopek. Drugače

pa pričnemo z iskanjem ostalih skupin, ki izražajo sosednje simbole na isti vrstici. To seveda opravimo v obe smeri (horizontalno in vertikalno), saj smer pisave še ni trenutno znana. Po pridobitvi obeh skupin, lahko prek razmikov med simboli določimo, katera najverjetneje diktira smer. Glede na rezultat, ter implementacijo, se lahko prične iskanje paralelnih vrstic za pridobitev delenega ali celotnega teksta. Te, pridobljene segmentirane vrstice, porazdelimo eno po eno v posamezne znake. Ti se zdaj lahko sestavijo v katerokoli reprezentacijo vsebine, ki nam najbolj ustreza. Najpogosteje je to konkatenacija vrstic, saj jeziki ki jih obravnavamo, nimajo vezejem, ter zato zahtevajo ta pristop, za razdelitev stavkov v korekne besede.

Ti posamezni znaki nam tudi omogočajo možnost natačnejših transkripcij, saj zdaj lahko hkrati uporabimo več različnih metod ponujenih prek OCR sistemov, tako da prek spememb reprezentacij vsebin, ugotovimo točne izpise za vsak znak, ter poravnamo transcripcije metod glede na konkretne simbole. To nam omogoči natačno znanje vsakega znaka v prizoru ter ne le grobe transkripcije celotnega besedila. S tem znanjem lahko anotiramo točne besede v vsebini.

V primerih kjer določene znake OCR sistem ni zmožen prepoznati z dovolj visoko prepričanostjo, lahko individualen znak obravnavamo še z drugimi orodji.

V zadjih poglavljih tega projekta je opisan še konkreten primer aplikacije, ki vsebuje vse našteto, ter prikazuje celotno moč segmentacije in preoblikovanja, prek uporabe slovarjev, pogostosti pojavov besed ter izdelave posebnih modelov za prepoznavo individualnih znakov, za pridobitev natančnejših rezultatov.

Dodatno se tudi omeni, da pristop iskanja znakov kot je opisan ni popolen, ter zahteva morebitno pomoč s strani uporabnika za najpoljše izide. Poleg tega pa so tudi omenjene ovire na katere lahko naletimo, ter načini kako jih najbolje razrešimo, kot so posebni dodatni pristopi za obravnavanje črno-belih slik, ter vsebin kjer barvne variacije lahko povzročijo težave.

7 References

[1] *MeCab*, https://taku910.github.io/mecab/. (Viewed on: 14/5/2020.) *(Cited on page 14.)*

[2] *UniDic*, https://unidic.ninjal.ac.jp/. (Viewed on: 12/8/2020.) *(Cited on page 14.)*

[3] *4-corner*, https://www.edrdg.org/wwwjdic/FOURCORNER.html. (Viewed on: 12/8/2020.) *(Cited on page 14.)*

[4] *EDICT-JMDict*, https://www.edrdg.org/wiki/index.php/Main_Page. (Viewed on: 12/8/2020.) *(Cited on page 14.)*

[5] *Sanseido*, https://www.sanseido.biz/. (Viewed on: 12/8/2020.) *(Cited on page 14.)*

[6] *K-means++*, https://en.wikipedia.org/wiki/K-means%2B%2B. (Viewed on: 12/8/2020.) *(Cited on page 4.)*

[7] *OpenCV*, https://opencv.org/. (Viewed on: 19/3/2020.) *(Cited on page 16.)*

[8] *Tesseract*, https://github.com/tesseract-ocr/tesseract. (Viewed on: 2/5/2020.) *(Cited on pages 1 and 14.)*

[9] *Lanczos resampling*, https://en.wikipedia.org/wiki/Lanczos_resampling. (Viewed on: 12/8/2020.) *(Cited on page 10.)*

[10] *A**, https://en.wikipedia.org/wiki/A*_search_algorithm. (Viewed on: 2/5/2020.) *(Cited on page 12.)*

[11] *Tensorflow*, https://www.tensorflow.org/. (Viewed on: 23/4/2020.) *(Cited on page 14.)*

[12] *Keras*, https://keras.io/. (Viewed on: 23/4/2020.) *(Cited on page 14.)*

[13] *Dataset, Results and Examples*, https://gitlab.com/JJS-Projects/ceg. (Viewed on: 17/8/2020.) *(Cited on pages 14, 16, and 17.)*