Collaborative OCR for Tabularized Data

Semester Thesis

Jan Bernegger

janbe@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Jochen Seidel, Tobias Langner
Prof. Dr. Roger Wattenhofer

March 18, 2013

Acknowledgements

I thank my two supervising assistants Jochen Seidel and Tobias Langner for their active help and many advices they gave concerning this project. I would also like to thank my girlfriend, my family and my friends for supporting me during the time I conducted this project.

Abstract

The Android Smart Shopping project was initiated to provide a tool to help users keeping track of their bought items by analysing receipts. As the used optical character recognition (OCR) engine Tesseract has not been designed for analysing photographs of receipts, it performs rather badly.

The target of this project is to provide solutions to improve the word recognition rate of Tesseract for receipts. The improvement consists of pre-processing the input image, identifying the individual fields on the receipt table and training Tesseract specifically for recognising the content of receipts. For this purpose I developed several image processing methods which are used in a client-server application, implemented as an Android app and a Java servlet.

Keywords: OCR, Tesseract, Image processing

Contents

Acknowledgements i

Abstract ii

1 Introduction 1

1.1 Description 1

1.2 Outline 2

2 Image Processing 3

2.1 Choice of the image processing tool 4

2.2 Correction of the receipt's rotation 4

2.2.1 Existing tools 5

2.2.2 Rotational error correction algorithm 5

2.2.3 Issues with the rotation 6

2.3 Histogram equalization 6

2.4 Line detection 7

2.5 Table element detection 9

3 Android Application 12

3.1 Porting the image processing to Android 12

3.2 Challenges of Android programming 12

3.3 Conversion IplImage to Bitmap 13

3.4 Rotation 13

3.5 Pixel access 13

4 Client-Server Application 15

4.1 Java servlet 15

4.1.1 JSON 15

4.1.2 Packets 16

4.2 Tesseract 16
4.2.1 Confidence - hOCR 16
4.2.2 Language generation 16
4.2.3 Training Tesseract 17
4.2.4 Levenshtein distance 17

5 Future Work and Conclusion 19
5.1 Future Work 19
5.2 Conclusion 20

Bibliography 21

A Android App Screenshots A-1

Chapter 1

Introduction

1.1 Description

Although we live in a digital world, receipts from the grocery store are usually still printed on paper, which makes it a time-consuming task to get an overview of all bought items at the end of the month. A program which would automate the process of scanning the items bought as well as archiving and analysing them would greatly simplify this task.The fact that a large percentage of the population uses smartphones nowadays makes them the perfect platform for deploying such a tool.

Keeping track of your bought items can bring various advantages. The obtained data can be used to get an overview on how much money you spend on a particular group of items or it can even be used to predict the entries of your shopping cart on the next visit to the grocery store. For this purpose the Android Smart Shopping project has been initiated at the ETH in Zürich.

However, it is necessary to get the required data first. As receipts exist for the very purpose of providing this data and are usually formatted similarly, the idea to scan and process these comes to mind. This can be achieved by using one of the numerous optical character recognition (OCR) tools available on the internet. This project uses the open source software Tesseract, originally developed by Hewlett Packard and funded by Google since 2006 [1].

As the quality of taken images of receipts to be analysed usually is bad due to the source of the image being a mobile phone's camera and the printing quality of the receipts themselves, it is necessary to prepare the image before passing it on to Tesseract. This project aims at providing solutions to achieve reasonable results from the OCR engine by the use of image processing methods as well as training Tesseract specifically to be able to recognise receipt content.

1.2 Outline

The paper is divided into the four chapters Image Processing, Android Application, Client-Server Application and Future Work and Conclusion. In the next Chapter I explain the various processing steps I developed from the raw input image to its detected receipt table which are used as input for Tesseract. In Chapter 3 the development of the Android application and the challenges I have been confronted with are described. Chapter 4 addresses the handling of the client-server application as well as the topic of how to train Tesseract. Finally, Chapter 5 gives an outlook on what additional work can be done and provides a conclusion for the project.

Chapter 2

Image Processing

OCR is a common method for digitizing printed documents. However, its recognition error rate heavily depends on the quality of the input image. To achieve the best results the input image should be a well oriented scanned document in high resolution. In order to be able to process receipts, which are usually themselves printed in bad quality, photographed by a mobile phone's camera, it is essential to pre-process the input image.

Tesseract is a pure OCR engine and is not designed to do various image processing tasks to improve the readability of input images. However, increasing the quality of the input picture and removing unimportant parts of the image greatly improves the obtained OCR results. Therefore, it is essential to find a suitable image processing library.

With the help of this library, the input image will be improved by applying the following steps:

Correction of rotation The other pre-processing steps rely on the input image being rotated correctly. Therefore, the first step has to be correcting the rotation of the image so the receipt faces top.

Finding the receipt table The next step is to find the table on the receipt which contains the article names, quantities and numbers and crop the input image accordingly.

Finding individual fields The final step is to find the lines and columns on the receipt table to determine all individual fields in the tablein order to process them separately

Among other steps which could also help to improve the input image quality (see Chapter 5 Future Work and Conclusion), these are the steps being discussed in this paper.

2.1 Choice of the image processing tool

At the time of conducting this project most C++ libraries still outperformed native Java libraries in both speed and functionality. As i did not want to pass on neither speed and functionality nor the simplicity of Java programs, I decided to use a C++ image processing library with a good Java wrapper.

After evaluating a couple of such image processing libraries, I decided to use OpenCV. It has been specifically designed for the very purpose of providing a reliable tool set of methods concerning Computer Vision. It is a cross-platform, open source library which focuses mainly on real-time image processing [2].

As a wrapper, I used JavaCV. Although it lacks of documentation and tutorials, the documentation of OpenCV is very reliable and can easily be adopted for JavaCV, which uses naming for most of its methods similar to OpenCV's C interface.

2.2 Correction of the receipt's rotation

Pictures of a receipt that are taken by a mobile phone's camera are unlikely to have the exact same rotation regarding the borders of the image (an example can be found in Figure 2.1). Therefore, the first step to improve the accuracy of Tesseract has to be to correctly rotate the input image.When the lines of text are not exactly parallel to the horizontal axis, even small angles can lead to many misinterpretations of its characters.

Figure 2.1: Picture of a rotated receipt

2.2.1 Existing tools

There are various tools which try to achieve such a functionality. However, all freely available tools I tested were only able to rotate the image up to an angle of approximately 10 degrees and failed to correct the rotation if the angle was higher.

2.2.2 Rotational error correction algorithm

The first step in the algorithm I developed is to use the Hough Transform of the image to find black lines and analyzing their angles relative to the horizon.

The Hough Transform [3] is a filter which extracts the features of a given image. To find straight lines in a picture, the lines, which can be described parametrically as $y = mx + b$, are geographically plotted in the two dimensional parameter space by the parameters r and θ, which are polar representations of the parameters m and b. Therefore, each point on the parameter space represents an instance of the searched object.

Applying the Hough transform to a picture of a receipt, its purpose is to find its separate lines, containing articles, quantities and prices.

The JavaCV implements the transform in the function `cvHoughLines2(...)`, which returns a stack of the lines found.The received lines vary regarding their angle. Furthermore, one has to take into account that the whole input image is examined, which leads to falsely detected lines with arbitrary angles. To select the final angle which is used on the rotation, the angles of the detected lines are split up into intervals of one degree. Finally, the center of the interval which occurred most often is selected.

Algorithm 1: Derotating algorithm

```
1 lines = HoughTransformation( image );
2 numberOfLines = new array(360);
3 for l = lines.pop() do
4     degree = Math.round( l.degree() );
5     numberOfLines[degree] += 1;
6 end
7 Select degree with highest number
```

However, the analysed receipts often show as much space between two characters of a word as between two consecutive lines. This leads to the Hough transform method often detecting vertical lines, consisting of individual letters of items standing below each other. An example of this can be seen in Figure 2.2.

Nevertheless, the developed algorithm is able to detect errors of small degree

in the rotation of the receipts and the final rotation in such cases is just shifted by $N * 90\,^{\circ}$. To correct this, a dialog in the android application was inserted, giving the user the opportunity to rotate the picture manually by steps of 90°.

Figure 2.2: Hough transform of a pre-edited receipt

2.2.3 Issues with the rotation

The rotation of the receipt at the end of the algorithm is done within the same image borders. Therefore, some small parts can be cut out of the image. This issue could be easily fixed by expanding the image borders before rotating it. However, this was not implemented due to the fact that this issue turned out to not influence the further analysis at all and it would only slow down the execution speed of the whole program because more pixels would need to be processed.

2.3 Histogram equalization

To further improve the character recognition, the receipt image was intended to be filtered with a Histogram equalization method before being sent to Tesseract. Histogram equalization is a technique to adjust the contrast of an image based on its histogram. Here, the transform is a geographical representation of the distribution of an image's brightness values [4]. The method redistributes the intensities on the histogram so the resulting distribution increases linearly with rising intensities, therefore resulting in an image with higher global contrast.

This method works especially well if both back- and foreground of an image are bright or dark.

However, instead of improving the character recognition, the equalization caused the output of the OCR software to stay the same or even worsen. A possible reason for this could be that applying histogram equalization may increase the background noise. This is why I dropped this attempt for improving the OCR output.

2.4 Line detection

After the image of the receipt has been rotated correctly, the next task is to find the table with the article names, quantities and prices and therefore to filter out the background parts.

Several steps have to be performed to achieve this. First, a row histogram is calculated over the lines. The row histogram used here is a curve where x represents the number of the line examined and y the number of pixels, whose black values exceed a certain threshold (see Algorithm 2). Before continuing, the curve has to be run through a low-pass filter to counter the high noise level caused by the parts of the image which do not show the receipt. The discrete filter makes use of the sliding window principle and works by setting the array value at the center of the window to the average value of all entries within the window.

Algorithm 2: Black and white thresholding

```
1  for int x=0; x<src.width(); x++ do
2  |  for int y=0; y<src.height(); y++ do
3  |  |  pixel = getPixel( x , y );
4  |  |  if temp<treshold then
5  |  |  |  lines[y] += 1;
6  |  |  else
7  |  |  |  lines[y] += 0;
8  |  |  end
9  |  end
10 end
```

Then a method to find the minima and maxima of the curve is used. Minimum-Maximum pairs which differ on less than one percent of the image's width are filtered out to reduce the influence of noise. The row histogram includes periodic minima, representing the spaces between the lines for the articles, their distance shows little variety (see Figure 2.3). The periodicity of the minima is used to determine the vertical end and beginning of the receipt table.

Assuming the receipt is centred in the image to be processed, the algorithm

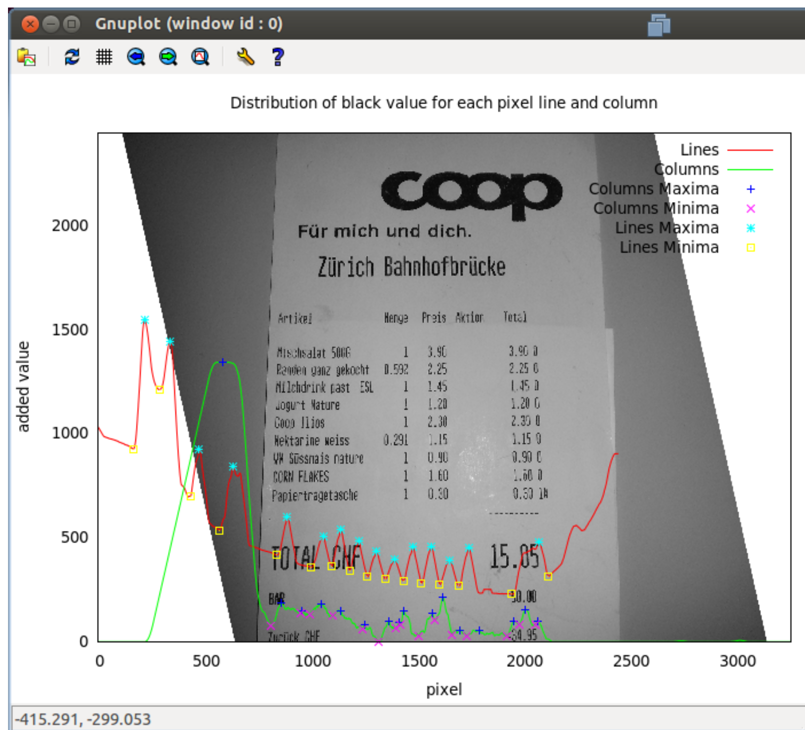


Figure 2.3: Minima and maxima of row and column histograms

picks the minimum at the middle of all detected minima. From this point, it checks each neighbouring minimum to the top and bottom if their distance to the examined minimum exceeds a threshold. To get this threshold, the median[1] of the distances of each neighbouring minima is calculated and multiplied by 1.5, to filter out small deviations in the distances.

If a neighbouring minimum exceeds the threshold it is considered as the end or beginning of the receipt table respectively. The according algorithm can be found in Algorithm 3.

A sub-image is then generated, cutting off the parts above and below the recognised table edges. Using this new image, the same thresholding, low-pass and minma-maxima filter algorithms are then applied to the columns of the picture, but using a higher tolerance value to fight background noise. The first and last minimum of the added black values of each pixel of the columns, representing the edges on the (horizontal) x-axis of the receipt table, are used to define the borders of the receipt.

[1]The median is used instead of the mean because the desired value should be near the real average of the neighbouring minima representing the lines. For the algorithm has to deal with all minima of the image and not only the minima representing the spaces between lines, the median filters out too high and too low values better than the mean, resulting in a better approximation of the real value.

Algorithm 3: Vertical table detection

```
1  colMed = calculateMedian( minima );
2  tolerance = colMed * 1.5;
3  startInd = 0;
4  endInd = minima.size()-1;
5  if minima.size()>2 then
6      lex = minima.get(minima.size()/2);
7      for i=minima.size()/2+1; i<minima.size(); i++ do
8          ex = minima.get(i);
9          if ex.getPixel()-lex.getPixel()>tolerance then
10             endInd = i;
11             break;
12         end
13         lex=ex;
14     end
15     lex = minima.get(minima.size()/2+1);
16     for i=minima.size()/2; i>=0; i– do
17         ex = minima.get(i);
18         if lex.getPixel()-ex.getPixel()>tolerance then
19             startInd = i;
20             break;
21         end
22         lex=ex;
23     end
24 end
25 start = minima.get(startInd).getPixel();
26 end = minima.get(endInd).getPixel();
```

2.5 Table element detection

The results of the previously applied algorithm are saved in a class containing the table dimensions and elements. To get the individual table elements such as article names, quantities and prices, the separating spaces between the elements have to be found. For this task, a function whose purpose it is to find major minima and maxima in a curve, developed in collaboration with my supervisor Tobias Langner, is used.

Said Algorithm 4 has two modes, each one searching for either a minimum or maximum. While the minimum search mode is active, it takes the lowest value as minimum. When the examined value exceeds the number of three times the last minimum, the algorithm goes into the maximum search mode, where it takes the highest value as maximum, until the examined value drops below the found

maximum divided by three. As soon as this occurs, the algorithm switches the mode again. This process can be seen in Figure 2.4.

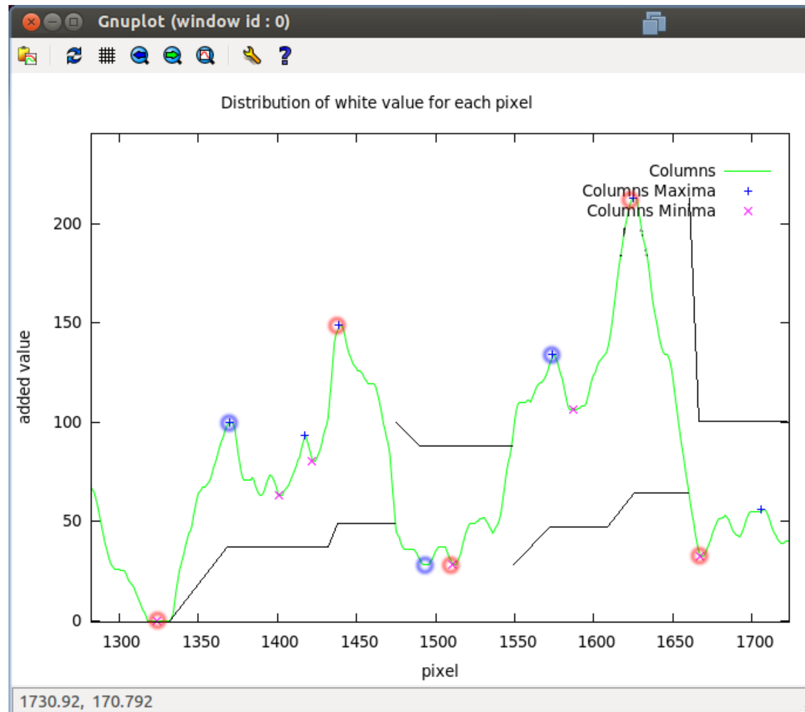


Figure 2.4: Major minima and maxima searching algorithm. The blue marked extrema are temporary, the red marked ones are the major extrema found by the algorithm. The black line represents an approximation of the limit for switching between minima and maxima search mode.

The minima found by this method represent the separators for the individual elements in one line. All individual elements of the receipt table are now separately accessible.

Algorithm 4: Major extrema detection

```
1  ascending = true;
2  max = Integer.MIN_VALUE;
3  min = Integer.MAX_VALUE;
4  xMax,xMin = 0;
5  result = new Vector<Extrema>();
6  for  i = 0; i <data.length; i++ do
7  |   int value = data[i];
8  |   if value >max then
9  |   |   xMax = i;
10 |   |   max = value;
11 |   end
12 |   if value <min then
13 |   |   xMin = i;
14 |   |   min = value;
15 |   end
16 |   if !ascending && value >3*min then
17 |   |   ascending = true;
18 |   |   max = value;
19 |   |   xMax = i;
20 |   |   result.add(new Extrema(MINIMUM, xMin, min));
21 |   end
22 |   if ascending && value <max/3 then
23 |   |   ascending = false;
24 |   |   min = value;
25 |   |   xMin = i;
26 |   end
27 end
```

Chapter 3

Android Application

To provide a convenient way to collect and analyse receipts, a client application for mobile phones operating with Android has been written. It allows users to take a photograph and analyse the picture on their phones before sending it to a server, which automatically responds with the results of Tesseract. A couple of screenshots of the Android app can be found in Appendix A.

3.1 Porting the image processing to Android

All the image processing code has been originally written in Java and later ported to Android. Although the Syntax is the same, multiple missing dependencies on Android had to be resolved.

While there are some limitations to its functionality, there are JavaCV binaries available for Android, though at the moment exclusively for ARM processors.

Due to higher availability of both computational power and memory of desktop computers, the image processing code was not designed for devices with restrictions in this area. Therefore, the developed code had to be optimised substantially, especially in terms of memory consumption, due to the huge memory requirements for Bitmaps which take approximately 32MB for a photograph with 8 mega pixel resolution.

3.2 Challenges of Android programming

Unlike regular computer programs, Android apps which are moved to the background by going to the home screen or opening another app are not guaranteed to stay in the memory of the mobile phone. Therefore it is possible that some or all variables previously calculated and stored in the app are deleted and set to null after an incoming call or a completed request to take a photograph. However, there are functions to handle this issue, `onSaveInstanceState(Bundle)`

and within `onCreate(Bundle)`, where the programmer is able to manually store and reload important values for the app respectively.

Threads are also handled differently on Android. While in regular computer programs without multi-threading the main thread can be used for all sorts of computations, this is not possible on Android. Here, the main thread represents the user interface thread (UI) and should not contain any heavy computations in order to prevent high latencies on the UI. Therefore computations such as the analysis of the receipt and networking code have to be executed in separate threads, since Android version 3.0 and 4.0 this is even forced by the compiler. In order not to be obliged to create and manage a thread pool for each written app, the very easy to use Android class AsyncTask is provided by the official Android SDK.

3.3 Conversion IplImage to Bitmap

The OpenCV image format IplImage provides a built in conversion to the BufferedImage format used in Java. For Androids equivalent Bitmap, however, there is no such conversion implemented. Therefore, to convert between the two image formats, the most important thing is to use the same format to store the pixel information for both image formats. Here, `ARGB_8888` for Android Bitmap and `IPL_DEPTH_8U` for IplImage are used, where each channel is stored with 8 bit precision. After two objects have been created with the same pixel format, the conversion can be done by copying the raw byte data of the pixels from one image object to the other.

3.4 Rotation

In the Android app, the previously developed rotational correction algorithm is used. To get rid of the possibly remaining error of $N * 90°$, a dialog, containing the corrected image as well as the three buttons "Rotate right", "Rotate left" and "Confirm", is shown to the user to enable him to correct the remaining error manually. After the user corrects the error and presses the confirm button, the analysis of the receipt table begins.

3.5 Pixel access

The functions implemented by OpenCV to access single pixels of an image are rather slow. The complete analysis of the receipt table, where the method has to be called up to twice the number of pixels in the source image, initially took

a *Samsung Galaxy S3* two to three minutes. Therefore, it was essential to speed up the pixel access.

The fastest way to do this is to access each pixel right from the raw image data. To get the pixel at the position (x,y), the byte array of the image `scrBuf` has to be accessed at the following position [5]:

```
srcBuf.get(y*src.widthStep()+src.nChannels()*x+n)&0xFF;
```

In order to only consider the last eight bits of the returned value `&0xFF` is applied.

The improved pixel access was able to speed up the complete analysis of the receipt table to approximately 30 seconds on the same mobile phone, which is an increase of speed by approximately 90%.

Chapter 4

Client-Server Application

Tesseract was designed as an OCR software to analyse printed documents. While it performs greatly on this task, the quality and properties of those scanned documents greatly differs to the ones of a receipt photographed by a mobile phone. Therefore, it is essential that the OCR software is able to learn from its mistakes and gets trained.

It is the advantage of a server handling the training data as well as the OCR that the whole community benefits instantly when individual users correct their wrong results. Also, it does not require the handling of very complex synchronization of the trained data between the devices of multiple users in order to provide good results.

4.1 Java servlet

Client serve communication requires well developed protocols. For this project a Java servlet is used, which is basically a web application running on a server responding to HTTP requests.

4.1.1 JSON

To be able to transfer objects from the Android app to the server and vice versa, JSON [6] is used. JSON is a standard for data exchange and it works by translating a Java object to a string which is used to generate the original object at the receiver. In this project the Android app sends objects to the servlet that are with encoded with JSON and encapsulated in a HTTP request. The servlet decodes the JSON object, processes the request and sends its answer back to the Android app, including its own objects encoded with JSON. While there are many freely available libraries for JSON, Google's implementation GSON [7] is used for its simplicity and because it works both on standard Java environments as well as under Android.

4.1.2 Packets

For every request, a packet class containing the objects to be sent as well as a communication identifier have been written. The communication identifier is needed for the servlet to be able to link a packet containing correction information with its preceding analysing request.

4.2 Tesseract

For this project, the open source OCR software Tesseract (version 3.01) was used.

4.2.1 Confidence - hOCR

Tesseract's standard output is the plain text it has recognised. However, it is possible to get a more detailed output, including the boxes in which Tesseract has found individual characters as well as the certainty with which Tesseract recognised each character. This information is given as a hOCR file which is an open standard for OCR output using HTML syntax. To parse the HTML formatted hOCR file, the free library jsoup [8] was used.

4.2.2 Language generation

Tesseract works with language files to provide a way of accurately recognizing characters for different languages. These files do not only include specific language data, but also the font data which Tesseract should be able to recognise later. For this project to work successfully, it is essential to provide such a language file for the specific font and language of the receipts, in this case mostly from Coop and Migros.

The process of generating such language files is a rather daunting task (see the tutorial [9]).It requires manually executing numerous steps, including one step where Tesseract tries to recognise the characters of a previously prepared text and the programmer then has to correct all the mistakes Tesseract made during this process. However, the open source python script *TesseractTrainer* [10] aims at automating this process for Tesseract 3.01. It requires a font and a text file as parameters, where the latter contains a large amount of sample text to automatically generate a valid Tesseract language file from scratch. As input I used a language file which can be found at Eugene Reimer's website[1].

[1] http://ereimer.net/programs/deu.SOURCEFILE

4.2.3 Training Tesseract

To improve the text recognition even further, it is possible to train Tesseract. Basically, the training data, consisting of image and box file pairs, is included in the language generation process to generate a new, improved language. These box files contain all recognised letters with their according coordinates on the image and are meant to be generated by Tesseract. However, Tesseract generates these box files with the characters it recognises, including the falsely recognised ones, and it is not easily possible to generate the box files for oneself, as the coordinates in the box files have to match exactly with those Tesseract would find.

Therefore, I decided to include only basic training in my server application, meaning that the language will only be improved if Tesseract makes small mistakes such as recognizing a few wrong characters and not recognizing something completely different. This has the useful side effect that images with very low quality where Tesseract cannot recognise anything are not taken into the language file, as this would most probably lessen the recognition rate.

After the Android app receives the recognised data from the server, a dialog on the Android app is shown to the user for each word where Tesseract hads reported a high uncertainty that the characters of a word have been recognised correctly. In this dialog, the original image with the text suggested by Tesseract are shown and the user has the possibility to correct the words (e.g. see Fig. 4.1). The corrected words are sent back to the server and analysed how they have changed. If the only changes are substitutions of some of the original letters, the server saves the image of the corrected field and the corrected text in form of a Tesseract box file to a specific correction data folder. This folder can then be regularly scanned by a modificated version of the Tesseract language generator script

4.2.4 Levenshtein distance

The changes from the recognised element to the corrected string are detected using a weighted Levenshtein distance [11] algorithm. The weights of both insertions and deletions are set to a certain threshold value, substitutions have a weight of one. The threshold represents the maximum number of corrected characters allowed for the reasons stated in section 4.2.3. After running the algorithm on both strings, its returned value is checked if it exceeds the threshold. If so, it means that either a deletion or insertion happened or more substitutions than allowed occurred and therefore, the correction data is ignored.

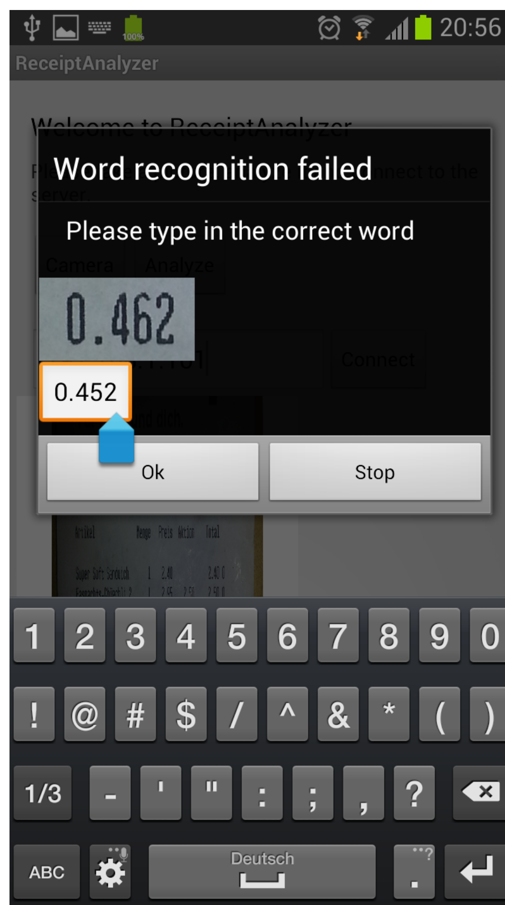


Figure 4.1: Correcting falsely recognised text

Chapter 5

Future Work and Conclusion

5.1 Future Work

Currently, the Android app gets the information if the photograph of the receipt could be processed by the server after having completed the whole analysis process. A more convenient way to tell the user to take another picture would be to analyse the picture taken on-the-fly and estimate a probability if it can be processed.

The rotational error correcting algorithm still counts on the final decision made by the user. An improvement here would be for the algorithm to find the correct rotation independently, which could be achieved by comparing patterns found in the thresholding process with previously defined ones.

Code optimization is very important for an Android application. Typical users do not expect high latencies while using apps, but the total latency of the analysing process is currently up to a minute with a high-end phone. For example removing the dependency of OpenCV in the project and implementing faster, on Android optimised methods for the image processing steps needed would greatly benefit the app's latency.

As the source of the receipt images usually is a phone's camera, the receipt can show distortion. However, the further image processing steps after correcting the rotational error require an image of the receipt which is not deformed. If this is not the case, the detected spaces between the receipt table's rows and columns may cut some of the characters in the more distorted part of the image. Therefore, providing a deskewing algorithm could increase the accuracy of the pre processing steps and, in connection with it, the OCR output itself.

There are many variables in the used algorithms which greatly influence their output. Currently, these variables are set to approximate values which I gained by testing a few samples of the respective inputs. However, for optimizing these values it would be beneficial to conduct an evaluation for every algorithm used.

On the server side, the character recognition could be improved by using

a separate Tesseract language containing only numbers for the receipt table columns for price and quantity. Furthermore, the language used to recognise the text part could be developed in a way which includes probabilities for the occurrence of a character. This would improve the recognition of characters with similar shapes but different probabilities, e.g. the characters `S` and `5`.

The word recognition could be further improved by comparing the recognised text with items users usually buy and correcting them if the their calculated Levenshtein distance has a low value.

5.2 Conclusion

The aim of this project was to improve the text recognition for photographed receipts for the OCR software Tesseract. The pre-processing methods developed for this purpose are able to automatically detect a receipt in an input image and divide it into separate single-lined fields of text which can be easily processed by Tesseract. I developed both an Android and Java Servlet application which handle the steps for preparing the image as well as giving the user the opportunity to correct the recognised words to enable Tesseract to learn from its mistakes. Although the final text recognition of the receipt table's fields is not yet perfect, it is overall a major improvement and a big step towards the desired functionality of a Smart Shopping application.

Bibliography

[1] Wikipedia: Tesseract. In: http://en.wikipedia.org/wiki/Tesseract_(software)

[2] Wikipedia: Opencv. In: http://en.wikipedia.org/wiki/OpenCV

[3] Wikipedia: Hough transform. In: http://en.wikipedia.org/wiki/Hough_transform

[4] Wikipedia: Histogram equalization. In: http://en.wikipedia.org/wiki/Histogram_equalization

[5] Google Groups: Pixel access from byte buffer. In: https://groups.google.com/forum/?fromgroups=#!topic/javacv/utLYEv8GAas

[6] Wikipedia: Json. In: http://en.wikipedia.org/wiki/JSON

[7] Google: Gson. In: http://code.google.com/p/google-gson/

[8] jsoup: jsoup. In: http://jsoup.org/

[9] Tesseract: Training tesseract 3. In: https://code.google.com/p/tesseract-ocr/wiki/TrainingTesseract3

[10] Rouberol, B.: Tesseract trainer script. In: https://github.com/BaltoRouberol/TesseractTrainer

[11] Wikipedia: Levenshtein distance. In: http://en.wikipedia.org/wiki/Levenshtein_distance

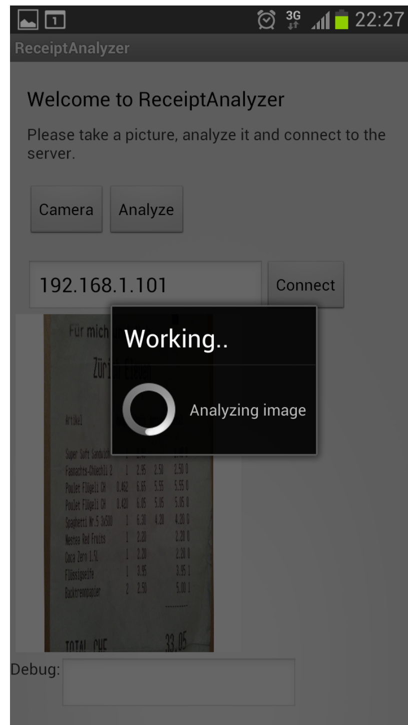
Appendix A

Android App Screenshots

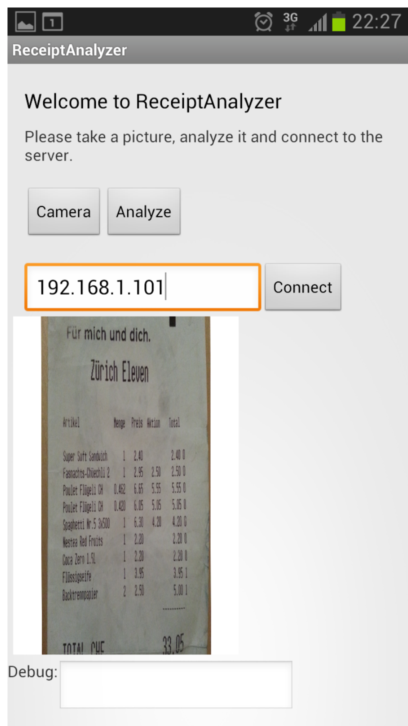
(a) Starting screen

(b) After taking a photograph

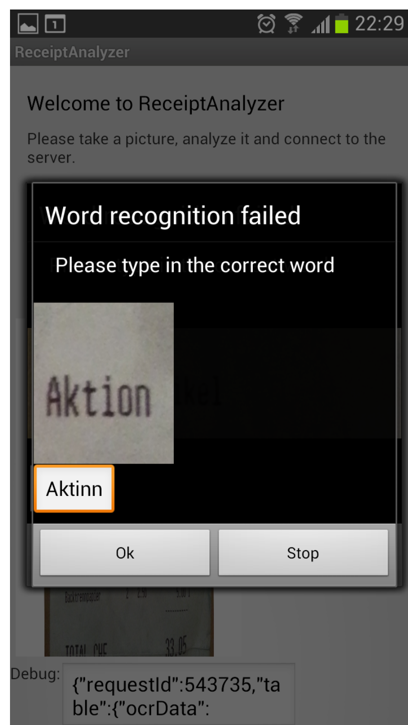
(c) Dialog after correcting the rotation



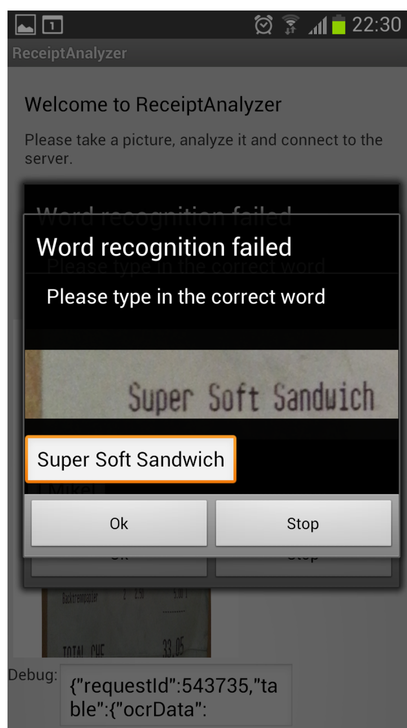
(d) Analyzing the image after confirming



(e) Screen before connecting to the server



(f) Falsely recognised word



(g) Correctly recognised word

(h) Presentation of the final result