Data Verity Mobile OCR Final Report

June 19th, 2018

Team Data Verity #2

Aimee True, Jack Pederson, Jim DeBlock, Quinn Tenorio
Client: David Flammer

Table of Contents

INTRODUCTION **2**
- Client Information 2
- Product Vision 2

REQUIREMENTS **3**
- Functional Requirements 3
- Non-Functional Requirements 3

SYSTEM ARCHITECTURE **3**

TECHNICAL DESIGN **5**
- Image Overlay 5
- Address Algorithm 7

DECISIONS **9**
- Guessing Fields and Text 9
- User Interface 10
- Usage of Code 10
- Optical Character Recognition Library 11

RESULTS **11**
- Performance Testing Results 11
- Summary of Testing 12
- Results of Usability Tests 12
- Features Not Implemented 13
- Future Work 13
- Lessons Learned 13

APPENDICES **14**
- Appendix A: Known Issues 14
- Appendix B: Original Mockup Design 15
- Appendix C: Final Design and Product 16
- Appendix D: Description of Algorithms 17
- Appendix E: Tesseract 19
- Appendix F: Fixing Bugs 20

INTRODUCTION

Client Information

Data Verity is a company based in Colorado that started in 2006. It was founded by brothers Gordon and David Flammer, with the idea of creating software solutions customized to their clients. They deal with developing cloud-based tools to meet various business needs of financial institutions. Some cloud based solutions that Data Verity has created over the years include full CRM enhancements, Learning Management, Problem Resolution, and Predictive Analytics. The project “Mobile OCR” was focused on the idea that Data Verity wanted a way for users, especially those on mobile devices, to be able to upload business card images and pull relevant information from it. Generally speaking, Data Verity is interested in information fields contained in the business card such as name, phone number, website, email address, etc. The project was to create an interface to provide some TLC when using an optical character recognition (OCR) technology like the open-source library Tesseract.

Product Vision

While the title of the project is “Mobile OCR” the actual result is a web application. Due to the necessity of functionality from this application, we were tasked with creating a web application that would also function as a mobile app further along its development. Thus, the final product is a web application that reads information from a business card file upload or camera photo. The application uses OCR technology to convert the readable data from the image file into editable text, which a user can alter or update. The application also assigns information from the business card to each specified data field using prediction strategies. The data fields accounted for are name, business name, address, email, web address, and phone number(s).

REQUIREMENTS

Functional Requirements

Using the Tesseract OCR library, an uploaded image of a business card is analyzed, and data of the image text is obtained. The program is able to take one side of a business card (a single image) and translate the words on the image into text. From there, our web application suggests predicted information to the user and visualizes how it corresponds to each field. The user can then edit that information. Finally, the user can verify that the

information is correct and an object containing the information is returned.

The English language is used as a base case for translation, since the English language is the only language required. The program is able to accept any conventional image format. Therefore, raster images such as jpeg/jpg, png, and gif, are all acceptable formats. However, one notable file format that Tesseract does not accept is pdf. File size is limited, and issues with this are outlined in Appendix A. The project was created using a Linux operating system.

Non-Functional Requirements

The product should handle information such as an image file, convert it into a text-based object, and securely store the information in an object which is returned. The object does not need to be stored anywhere after being returned. The application is accessible on the internet using a desktop computer with possible later implementation on a mobile application. Using the application also requires no prior knowledge or training. It mainly accepts business cards, but other documents could also be used (e.g. invoices) to copy information from.

SYSTEM ARCHITECTURE

Figure 1 outlines the system architecture of working on this program. The team primarily interacted with the client database, program, and OCR app, with no modification to the web server or internet. Data Verity provided us with an online database which was utilized to build the application. Within the database, our program was stored using ExtJS, Javascript, HTML, and CSS. The file upload portion of our program was provided to us within a form through the online database. Compilation of our program provides the user with an OCR application, through the use of the internet and a web server. The end result is the final product: a web application for optical character recognition of a business card where a user may actively edit the information pulled from the image. There was no implementation of any information being stored into the database at the time of project completion, as the client wanted to focus on the functionality of the application.

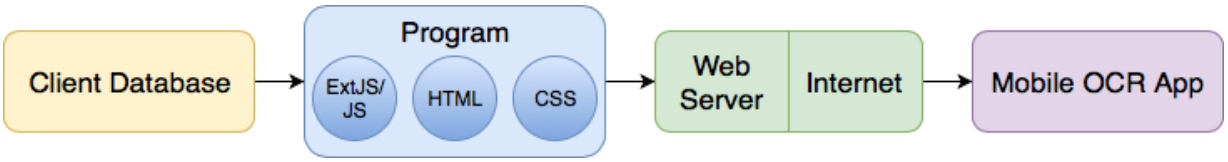

Figure 1. System Architecture

Figure 2 shows how our application works from a user perspective. First, the user utilizes a file upload function to upload a business card image of their choosing. The image is then displayed with a series of colored boxes. Each box, referred to as a "field box," contains an input field along with a clickable button. Then, the optical recognition software, Tesseract, is run. Depending on the image, Tesseract could complete its scan in a matter of seconds, or may take longer. Because of this, until Tesseract finishes running, the user is unable to do anything besides look at the business card image. The input fields in the field boxes may be filled in, depending on the image and Tesseract's recognition capabilities. Sometimes every field will be filled and sometimes only one or two may be filled. This is a result of applying a confidence level to each word, then removing words with low confidence from the object output by Tesseract. The user can then examine the field boxes and see if the information provided matches what is on the business card. If the user is unsatisfied, they can then edit the information in the input fields, if for example Tesseract reads a word incorrectly. They can also add additional information from the business card that might not have been added. This is done by clicking on the button in the field box, and then clicking on the highlighted words in the image. Re-clicking the button stops the user from selecting words. When the user is satisfied with the information, they can submit the information by clicking an "Okay" button in the bottom-right corner of the screen. Submittal returns an object containing the confirmed information from the UI. The object is not stored anywhere, as that implementation will be included when the mobile application aspect is implemented.

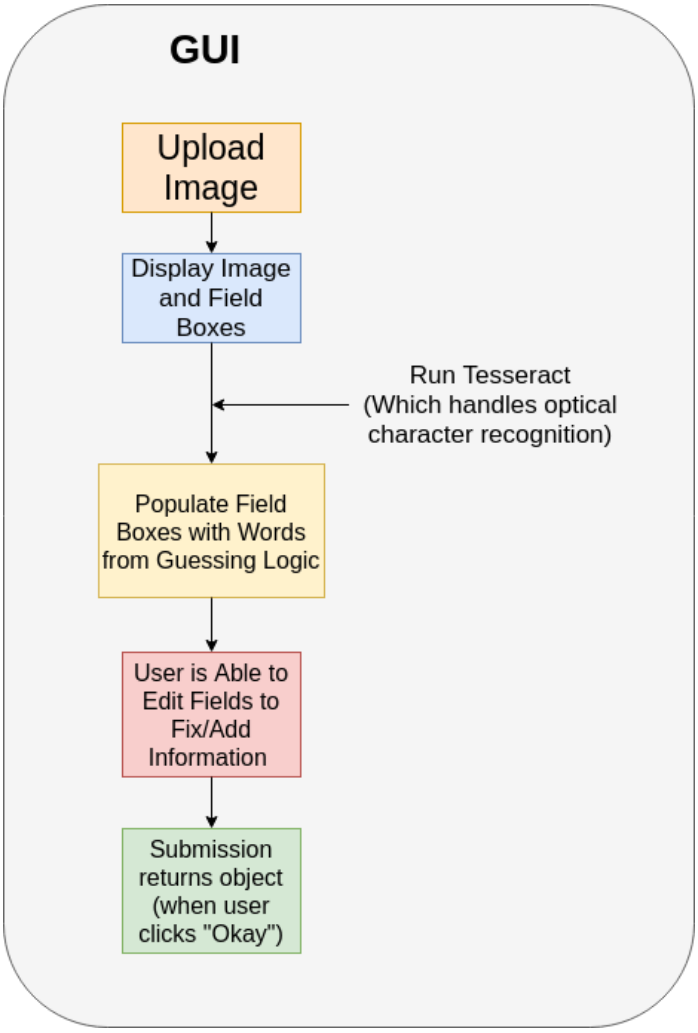

Figure 2. High-level design of UI

TECHNICAL DESIGN

Our product can be divided up into three main components: Tesseract, the user interface, and the prediction algorithms. Sponsored by Google, Tesseract is a free software providing optical character recognition, discussed in Appendix E. The user interface includes the image, image overlay, and field boxes. The field boxes allow the user to select information corresponding to certain categories, such as name or phone number. The prediction algorithms consist of the algorithms used to guess what text on the business card matches what fields. More information on prediction algorithms besides the address can be found in Appendix D.

Image Overlay

The user interface consists of two main components: the image itself and field boxes. When the user clicks on the button in a field box, an overlay is shown over all of the words in the image. Some words might not always have an overlay, as that is dependent on Tesseract and how well it detected the words. Generally though, most words on the business card will have an overlay. Noted in Figure 3, once Tesseract has been run and the prediction algorithms completed, the program creates the image overlays. Each field box has its own image overlay. For example, the 'Name' field box has its own image overlay which consists of its own set of HTML5 DIV elements. Each DIV element corresponds to a specific word on the image, and overlays the dimensions of that word on the image. So the DIV element is comprised of an area in the shape of a box with an overlay element that has opacity so the text underneath the DIV element can be seen by the user.

Typically, the color for overlays is gray. However, there is a 2-D Array, called fieldWords, which contains the selected text for each field. A 2-D array is indexed by two subscripts, one for the row and one for the column. In the case of fieldWords, rows contain fields and columns contain words from the image associated with those fields. For each field image overlay, the words are gray unless a word has been associated with the current field that is being edited. If the word is associated with a field, the overlay color for that word corresponds to the associated field color. For example, for 'Street Address,' its field box is purple. If there is information in fieldWords corresponding to the street address, then the street address overlay has corresponding words in fieldWords which are purple. While an overlay is being displayed, a JavaScript onclick function is used. If a word is clicked on by the user, that word is added to the fieldWords 2-D array and its overlay color is changed. If the user clicks on that same word again, then the word is removed from the 2-D array and the overlay goes back to gray. To make the overlay appear for a specific field, a user simply has to click on the button in the corresponding field box. This button will show the overlay if it is not currently on screen and it will hide the overlay if it is already showing. This is done by using the element's CSS style.display attribute and setting it to 'block' or 'none'. 'Block' makes the element appear meanwhile 'none' makes it hidden. There is also a "Clear text overlay" button which sets the overlay for all fields to 'none.'

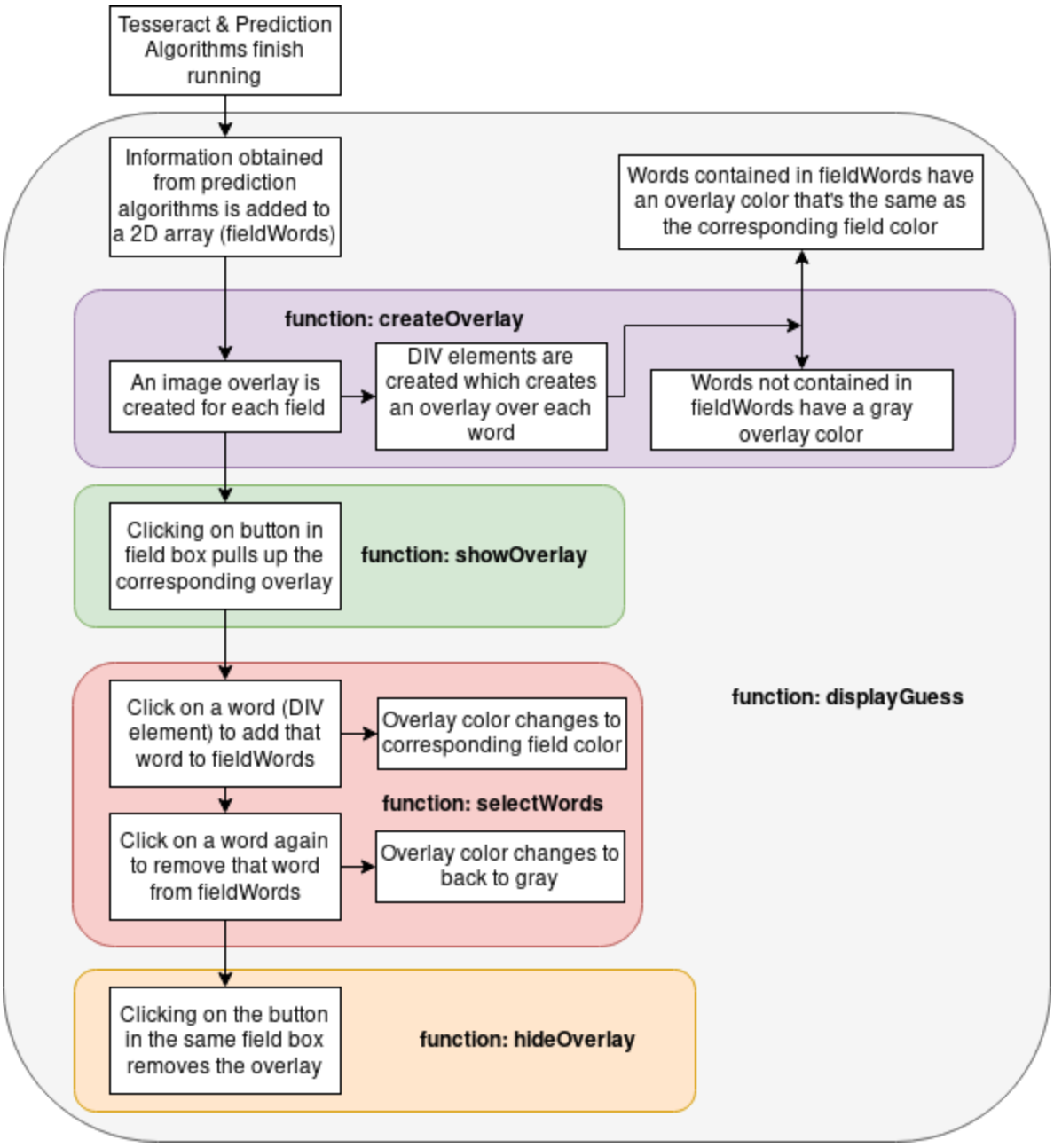

Figure 3. Flowchart of how application creates and uses image overlay

Address Algorithm

Generally, all prediction algorithms score the words within a line of words, based on how well they meet the criteria of what field they are checking for. After the loop, it checks each word for the highest score assigned, and assigns it to the corresponding field. The algorithms score the words separately, though some algorithms needed to check more than one word, so passing through the entire line was a better option for

correctly predicting text.

Focusing on one of the larger prediction algorithms, addresses was the most interesting. Addresses can be written in any number of ways, which makes detecting them quite challenging. Though, since each address consists of the street, city, state, and zip, it is possible to break up the algorithm into four steps. The first step in finding the address is to look for the zip code. For the most part, zip codes are 5 digit numbers. Sometimes zip codes are written with an additional 4 digit number, though we never encountered a business card with this. Our algorithm simply looks for a 5 digit number and also checks if it is at the end of the line, since zip codes are always at the end of the address.

The next step is to find the state. We originally attempted actions to check for two capital letters, or locate the word before the zip code to find the state. Both options would not account for states with two word names like New Jersey. Instead, we created an array of all 50 states and their two-letter state codes, and checked against that to find the state. The only issue with this method was if it did not match the array perfectly, it would not be recognized. This happened if Tesseract read the state wrong, or if the business card itself spelled the state wrong. So, as a backup, if it did not get the state, it would pull the word in front of the zip code. This still did not account for two word state names, but still provided a fix to the challenge with finding state

Next, it checks for the street address. This was the most challenging part, as there are so many ways to write them and they could be written on different lines. If they are on separate lines, the algorithm can pull the entire line before it, otherwise it has to dissect the line. All street addresses start with a number, so it searches the line for the first number that is not a zip code. From there, it searches the line until it finds a street ending or a number (like suite or apartment numbers). The main problem with this approach is the vast number of street endings that exist, as well as the number of ways to write them. This method addresses the largest challenge, and works a fair amount of the time.

Finally, the city is checked for. If the street address is on a separate line, the city is predicted to begin at the start of the line and end at the state. If they are on the same line, then the city begins after the street and ends at the state. This algorithm is not perfect, but addresses most challenges we faced in predicting street addresses.

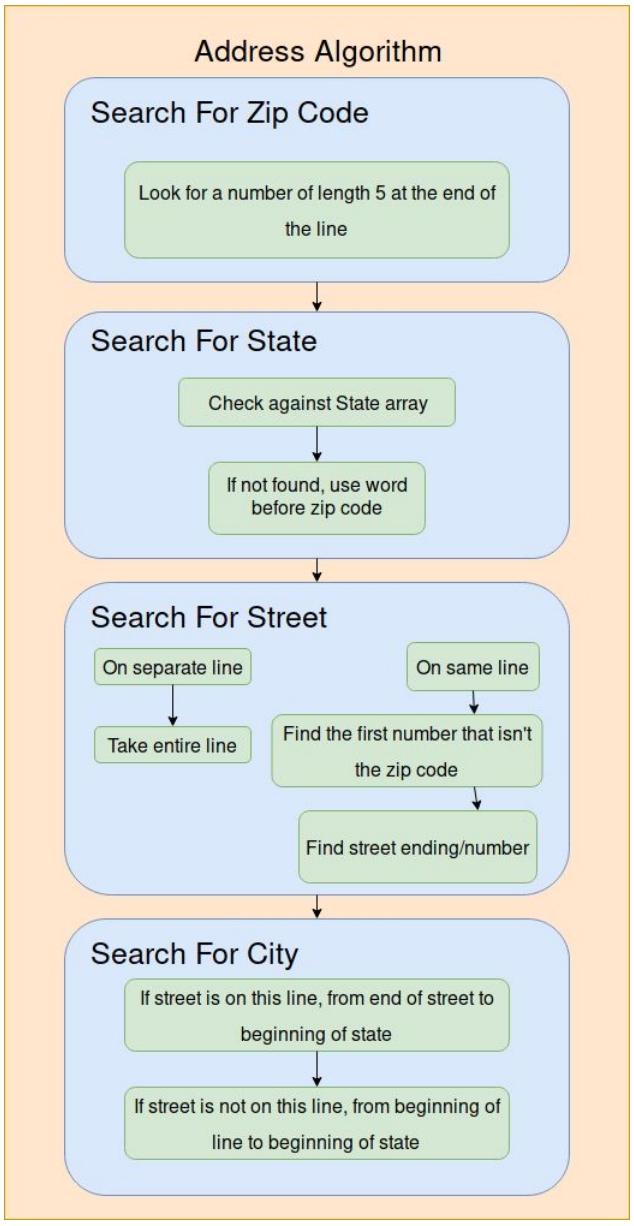

Figure 4. Address Fields

DECISIONS

Guessing Fields and Text

On a more general discussion of algorithms, we initially chose to loop through all words that Tesseract returned and guess which words corresponded to which fields. This approach only assigned fields if the algorithms recognized them and it only allowed the user to edit a field if it was assigned words. After a discussion and a demonstration with

our clients, we decided to implement a scoring system for each word. We changed our algorithms to take in one line of words at a time, so each word would be assigned a field type to it based on which one matched best. Then, for each field, the words assigned to that field depend on the score they were previously given. This approach would allow the user to edit every line, even if it was not recognized as any field type.

User Interface

On the front end, the graphical user interface (GUI or just UI) was a necessary component of the product. The UI changed as the project progressed, and the original UI design varied heavily from the final design. For an idea of how the UI evolved, an original mockup is included in Appendix B and the final product of our design is included in Appendix C. The design had to be suitable for a mobile platform, and usable on nearly any mobile device. The client wanted a field for the uploaded file to display, and another field where text information from the scanned image would be held. In addition, there is a button to show/clear the identified information and an input field for text where the user could edit the information from the document. After ensuring that all of the information accurately reflected that of the uploaded document, the user could hit a 'submit' button. This would return the information to a database as an object, to be dealt with after field session, as outlined by Data Verity. Overall, the UI we designed is clean, simple, and intuitive to use, with its operations made clear to the user simply by its buttons, input fields, and short text descriptions.

Usage of Code

In terms of actual code, our web application will be implemented as a mobile app later on. Thus using the HTML, CSS, and JavaScript made sense because all three are main programming languages of websites (both desktop and mobile). As a primary language, we chose to use JavaScript, as it is event-driven and can manipulate DOM elements. Our client Data Verity suggested that we also use ExtJS, which is a JavaScript application framework that they often uses for projects. The advantage of using ExtJS is that certain aspects of the project were much easier to create. On the other hand, when transferring the web form of our application into mobile form, some difficulties could arise. Mobile ExtJS sometimes uses slightly different syntax than normal ExtJS, unlike HTML, CSS, and JavaScript, which are all the same for web and mobile development. Because of this, our use of ExtJS was rather limited and we used normal JavaScript instead where applicable.

The decision to use normal JavaScript worked well with the OCR library that we chose to us. That OCR library was provided for us, but all other elements of the product had to be designed and built by us. Data Verity's database made this task easier, as it allowed us to

run demo's of our product as we were building it. Before we began working on our project, Travis Kopp implemented the functionality for uploading a file. Therefore, on our end, we had the image file and had to build everything from there. This included the user interface and other aspects of our code that the user does not see, such as the prediction algorithms. We had to deal with displaying the image, developing a user-friendly way for users to edit text/information, an intuitive way for users to select information from a business card, writing and implementing the prediction algorithms, and making everything work neatly together. We also had to extract information given to us by the OCR library we implemented. This involved learning how the OCR library worked and the best way to use it for the final product.

Optical Character Recognition Library

Rather than building and optical character recognition application, we utilized an open source OCR library to assist us. We were tasked with building an application that utilized an OCR framework to enable the application to use and manipulate text. As a result of us using ExtJS and Javascript, our client suggested the use of the Tesseract.js framework to accomplish the task of OCR. We decided to go on with this, as it would allow us to focus more on the application and algorithms, and less on the character recognition. More information on Tesseract and how it functions can be found in Appendix E.

RESULTS

Performance Testing Results

Tesseract is entirely self sufficient in producing text and data from an image. However, wait times exist due to Tesseract reading the image, especially as image size increases. This issue is out of our control, as Tesseract needs time to process the image. The methods that we wrote operate at a level that adds a negligible amount of wait time for the user. Performance was kept in mind as we wrote them, so that operation time did not increase as a result of our methods. How well our prediction algorithms work depends on how well Tesseract reads the image. Common mistakes that Tesseract makes are reading an '@' symbol as a copyright symbol and reading the letter i as the number 1. If there is information on a business card that corresponds with a certain field, it will be assigned to that field. However, name and business name might be incorrect, but that is often because of Tesseract misreading words.

A considerable level of abstraction was implemented in our code such that code length and performance speed were cut down significantly. For example, the function

evaluateWordForType() is a clean way of looking for the indicated text field type. This function calls the appropriate function to determine whether lines of text belong to a specific field type. Another instance of abstraction is in the function storeWordOrder(), which is used for two functions, selectWords() and displayGuess(). It takes in the text data given from Tesseract, then creates an array detailing the order of text found on the image file. Both functions that call storeWordOrder() previously had the same functional logic, but with different objects. This was unnecessary, as code should be as simple and non-repetitive as possible. Thus, the new function was created to pass in the objects, and deal with the data in the same way. We aimed to make our code as clean and effective as possible by introducing the concept of abstraction.

Summary of Testing

In examining performance, we tested the application as well. The application works well with major browsers (Google Chrome, Firefox, Microsoft Edge, and Opera). Internet Explorer was not tested due to its age and lack of availability on new Windows computers. For it to work completely with Chrome, users should navigate to 'Developer Tools' > 'Network' and select the checkbox 'Disable cache' so that any updates to the code carry through to the web page. This should not be an issue as users utilizing this application will be within the client's company.

The other main way that our product was tested was using over twenty business cards. The core functionality of allowing for alteration of text worked on every test. Our algorithms for guessing text beforehand to assign to types also worked a significant portion of the time. The small percentage of time that it does not work is a result of the OCR framework misreading text or a result of the business card format. Some other issues arose from image files being too small or too large. The former would prevent Tesseract from accurately reading the text, and the latter would cause Tesseract to crash. This could be an issue in the future if users create image files with high-quality cameras or scanners, and do not manually resize them.

Results of Usability Tests

Furthermore, in order to make sure that our software worked with different business cards, we searched the web for cards with different fonts, formatting, and sizes. We tested our algorithms on about 25 cards to ensure that we had accounted for a variety of possible layouts. Results acquired from these tests (i.e. one card having an address on one line as opposed to two) provided us with more data, which we were able to use to make our algorithms function optimally. Travis Kopp, our point of contact from Data Verity, also regularly accessed our code and tested it. As someone who was familiar with

our code but was not directly involved in writing it, Travis was able to find bugs and problems that we had not noticed. This allowed us to fix problems as they appeared, as Travis was consistently looking at our application and testing it.

Features Not Implemented

We were unable to implement a more involved artificial intelligence (AI) aspect, which could learn and be able to detect certain values and assign fields, such as phone numbers. The AI's ability to identify field types would increase the more business cards it came across. Our project, in its current state, can guess what information on a business card pertains to what fields, but it does not learn. In addition, there was not enough time for formatting the application to work as easily on a mobile device. Both of these features were discussed with the client and determined to be stretch goals to implement if time allowed.

Future Work

Both unimplemented features above can be completed in the future. Namely, creation of AI would make the program function more along the lines of what was sought after. In addition, implementing use of the returned object values can be completed. Currently, when a user submits (clicks "Okay"), an object is returned. Nothing is currently done with the object. We were given instructions not to worry about the object of values, as it would be dealt with at a later date after field session. An extension on the object could be that the information is then entered as a contact in an address book or is stored for later use. This would be useful if the mobile aspect of the project is implemented, as information would be easily stored on a mobile device. Another extension to implement would be the feature involving the

Another aspect that can be worked on is the issue of two-sided business cards. Currently, our implementation only allows a user to upload one side at a time, which can lead to incomplete data when dealing with two-sides. Being able to upload two images simultaneously, run Tesseract on both images, and use the UI to select information would be a worthwhile extension.

Other minor implementations also exist. For instance, being able to upload a document other than a business card, like an invoice, was discussed with our client. Also, the web application is only set-up to work for the English language. However, implementing other common languages such as Spanish or French could easily be programmed, as Tesseract already has the functionality for those languages.

Lessons Learned

Throughout the five weeks of working on this project, there were three main lessons that we learned, with a help from our clients and ourselves. First of all, JavaScript is asynchronous. Up until this point most of the code that we have used and been taught has been synchronous, where code runs logically and in order. Mainly writing in asynchronous code has been an adjustment, as we had to learn when our code would run as expected and when it would run out of order. However, because JavaScript is asynchronous it can be faster than writing in another language. In addition, the OCR framework that we used contributed to the difficulties of working with the code. Tesseract would run in the background while other code was dependent on Tesseract loading in the data read from a file, causing issues with code and its asynchronous nature.

In addition, using unique IDs is an important aspect when creating web apps. While it can be easy and simple to identify an element by an ID such as 'ComboBox,' it can cause large issues later on. If an element is identified without a unique ID and the web application is loaded and reloaded, information previously identified with 'ComboBox' will still be identified. If unique IDs are generated and added onto an identifier, then each time the web application loads, there should be a new ID for each instance of an element. For example, one instance the ID might be '123-ComboBox' and the next it might be '987-ComboBox'. This is how companies want their web apps to function, to ensure that each session of the application is independent of other sessions.

Lastly, accessing HTML elements can be costly in terms of time and computing power (especially if there are lots of them to look through). Working on this project, it became unruly to use the typical line 'document.getElementById()' to access an element and change styling or data. Thus, we learned to store frequently accessed elements in an object or array which could be referenced when we needed to change values, styles, or data. For instance, instead of using the above line of code to access an element, it could be referenced as so: 'this.domElements[0].style.color = white'. This was used several times throughout our code, namely for accessing divs, input fields, and buttons.

APPENDICES

Appendix A: Known Issues

Image Sizes

Tesseract.js can handle almost all image sizes, with the exception of very large images.

This will be an issue with images taken on a mobile device, as they (sometimes) can be too large. We found that the size of the image around or over 3500x2000px will not work with Tesseract. Currently, there is nothing that will stop a user from uploading an image of any size, and there is nothing that resizes the image object before passing it into Tesseract.

Address Algorithm

Sometimes the address is not the only thing on the line. For example, the line might not end with the zip code. This poses an issue, in which the address only gets recognized if it is the last word on the line. The street address uses some common street endings in order to find the endpoint, but not all street endings were accounted for.

Business Name Algorithm

Like the street address, a list of 'buzzwords' were created in order to more accurately choose business names. This is not the only way to detect business names, so if it does not contain a buzzword it can still be recognized, but this method is not perfect.

Name Algorithm

Because of the unique quality of names, there was little to go off of as far as buzzwords or indicators for a name field. Business cards usually do not have indication words that a name is after, unlike phone and fax numbers. Therefore it was necessary to use regular expressions to filter text through and ensure the only text fields being considered for name were only alphabetical. Sometimes, due to the nature of Tesseract, the text from a name is incorrect and the code does not consider it a candidate for name as a result.

Confidence Levels

There is a function named ignoreConfidence(), which will ignore confidence levels up to the value passed in. A good confidence level found to eliminate unnecessary details is 64. However, sometimes Tesseract will detect information needed at a very low confidence level (such as 30). When this happens, information that is needed and applicable, is removed. However, if the confidence level is lowered to 30, then extra things on a business card are read as words. For example, pictures and other small details are read in as text, which can cause the image overlay to be difficult to read.

Appendix B: Original Mockup Design

Originally, the design for this web application was to look like Figure 5. However, while some elements from the original mockup can are still present in the final design,

complications arose preventing the original design from working. Business cards usually have information clustered together, so putting text boxes below, above, or beside words would not work. Typically, a business card does not look like the one in Figure 5. To work around this, we then attempted overlaying the words on the image with text boxes. Then, another problem arose of not being able to see the text underneath and therefore made it difficult to edit information. Several ideas were played with to solve this issue, such as having a button to hide the text box overlays or having a duplicate image with no overlays. In the end however, it was decided that the best course of action would be to have the text boxes displayed beside the image instead of on top of it.

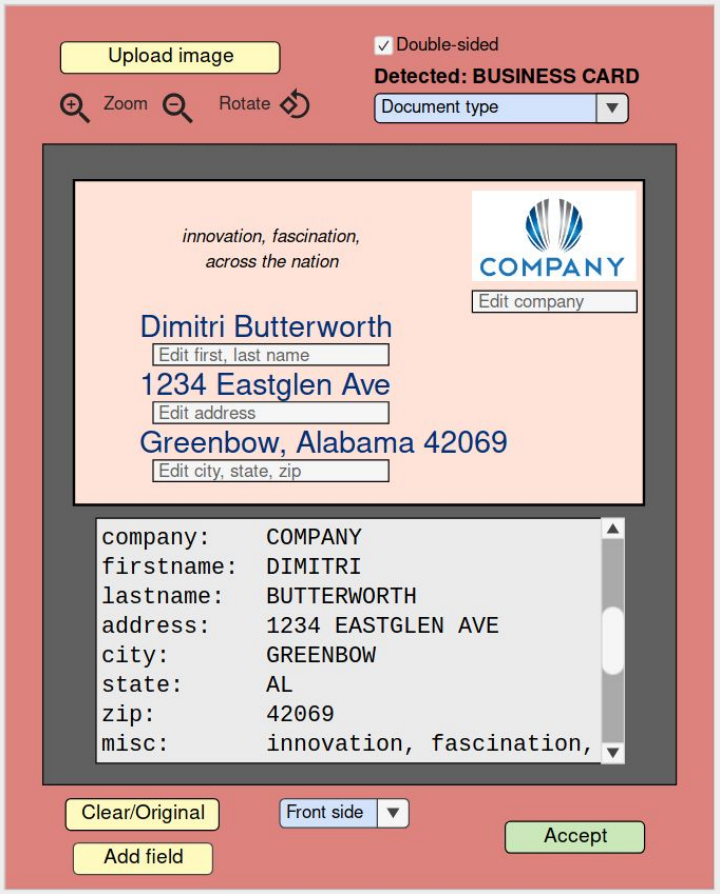

Figure 5. Original design mock-up

Appendix C: Final Design and Product

Figure 6 shows a basic final mockup of the design that was arrived at after trying out several different ideas. Figure 7 shows an example of the final product during a demonstration, after a business card image has been uploaded and Tesseract has successfully loaded.

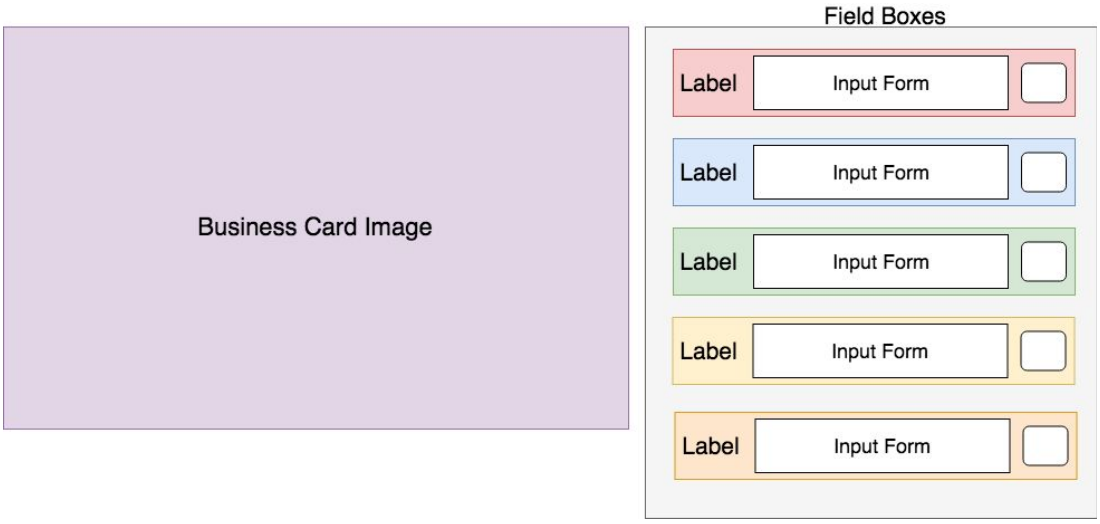

Figure 6. Final mockup design

Figure 7. Screenshot of final design/product

Appendix D: Description of Algorithms

Name Algorithm

The algorithm for searching and successfully finding the name of an employee on their business card takes in a line from the scanned text. The text is then separated by word to

evaluate whether the word string in question contains any characters other than capital or lowercase letters. After determining that a word contains no non-alphabetical characters, the word is added to an array, which is then parsed through and filtered through a regular expression that explicitly allows only [A-Z] and [a-z] ASCII values through. This is for redundancy in the case that a non-alphabetical character is not caught on the first try. In addition, the algorithm begins by parsing through each word from the card, and checks if there has already been a positive, non-zero score applied to the word. If there is a score attributed to that word and it is greater than zero, the algorithm stops and returns that word string. Finally, when a word has made it through that does not have a positive score, and contains only characters that fall into the ASCII value ranges specified above, a for loop parses through the line on which that word was found. Each word on that line is then attributed a positive score, so ensure that when selecting a name, each word in the name string is considered a name. This is to ensure that when the name field is populated, a single word such as first or last name does not appear by itself. Lastly, the name algorithm is implemented last in order of detection algorithms, to ensure that non-name strings do not get confused with a name, such as job titles and descriptive words.

Business Name Algorithm

The business name algorithm is one of the last algorithm that gets tested for, since it relies on checking to see if a that line that it is passed through already has a score assigned to it. If it does, it just ignores the line. Otherwise, it first checks against a list of buzzwords, like “incorporated” that are dead giveaways of it being a business name. After that, it checks to see if it is close to the top of the business card, since business names are typically listed towards the top.

Address Algorithm

See section Address Algorithm in Technical Design.

Phone Algorithm

The phone algorithm looks for some labels like “phone”, “mobile”, “fax”, etc. These will increase the score assigned, since they will most likely be followed by a phone number. Next, it parses through the line, looking at each individual character and counting the number of numeric/alpha characters on the line. If there are at least 10 numeric characters on the line, and the number of numeric characters is greater than the number of alpha characters, then it is assigned a high score, since it is assumed to be a phone number at that point. This algorithm already separates different phone numbers into an

array, so if there are multiple on one line it can detect all of them. It also checks to see if any phone numbers have already been assigned, and if so, it will separate the next numbers that it finds into the 'fax' category. This means that if there are more than 2 phone numbers then the last 2 will be considered fax. Unfortunately since there were only two fields for phone numbers, this was the only way to go about this.

Email Algorithm

The email algorithm takes in a line from the business card. It then iterates through each element or word in that line. For each word, a score is given based on how likely that word is an email address. The scoring is done subjectively. If a word contains an '@' symbol, then a word is given a high score. Some other elements that are used to identify an email are common email endings (e.g. 'com', 'net') and characters that are similar to the '@' symbol such as the copyright symbol. Elements such as 'www' decrease a word's score, as that word is probably a web address, not an email. This algorithm also checks to see if in the line passed in contains the word 'email' or a way to represent email such as just the letter 'e.' If so, then somewhere on that line probably contains an email, so for each word on that line, a small number of points are added to the score for a word.

Web Address Algorithm

The web address algorithms functions much like the email algorithm. It takes in a line and then iterates through all the words in that line, giving each word a score. If a word contains a common web address ending (e.g. 'com', 'net'), points are added to the score. If a word contains other common elements found in a web address, such as 'www' or 'http,' the score is also increased. However, an '@' symbol will decrease a word's score, as that word is likely an email.

Appendix E: Tesseract

Tesseract is a Javascript framework, making it applicable to fit well into the program. It can recognize several different image sources to pull information from. It is particularly useful as it separates text and its properties into easily accessible types and data. Tesseract returns an object with information from the image. Breaking down the object, there are several different ways to view the data, like separate words, lines, and even symbols to allow for a wide range of use. For each word and line, the confidence and its location on the image can be pulled. For instance, the location is stored in an object called 'bbox' which includes a beginning and ending location for the x and y coordinates of an element. This is especially useful for the client's application, so we may accurately allow any user to be able to decide which text can be assigned to any field. The usage of

Tesseract for OCR is exceptional at enabling the rest of the program to flow smoothly, particularly the aspect of guessing field type with text read from the image.

Appendix F: Fixing Bugs

Aa fair share of frustrations occurred during the testing process, as more and more functionality was added. It was not uncommon to have a piece of the code working one day, only to come back the next day to find it broken. These kinds of issues were resolved by using the web browser's developer tools, allowing us to see where things went wrong. By using this tool, we could also access the console log, which proved useful for printing error messages and benchmark statements to ensure the code was running through each for loop, case, and if-else statement correctly. It also allowed for us to see where in our code errors originated from. This do not always immediately identify the problem, but it was at least a step in the right direction. This method of combing through the code was ultimately the crutch of this project, and helped us out countless times when we were utterly lost on what was wrong.