Final Report

Team: JD Baugher, Austin Thompson, Tyler Thrailkill, Grant Walker
Client: Nimbee
06/13/13

Introduction

The members of the Boulder-based startup Nimbee have found great success in the educational field with their previous startup, Kerpoof. The now hugely successful website was created in 2006 to help children create and learn in a safe online environment. After two years and an incredible rise in monthly traffic, the website was purchased by Disney in 2009 and remains a popular tool for teachers all over the world. Despite their success, Nimbee presents yet another opportunity for the talented team members to create an award-winning educational tool.

Teachers need methods for real-time, interactive lectures which can be delivered on a variety of platforms, similar to Khan Academy instructional videos. Teaching fractions to middle school students is infamously difficult and the problem still does not have a satisfactory solution. Nimbee is focused upon solving this problem with their next product. They have begun work on an application called Woot Math, with a planned release on Web, iOS, and Android. Woot Math is designed to help teach students fractions and other mathematical concepts using tactile, visual feedback for each problem. Woot Math will not only help students by aiding in the visualization of fractions, it will also recognize and process their handwritten equations. Nimbee has requested that we assist them in their Woot Math project by developing a framework for saving and evaluating this handwritten user data.

Students need an intuitive method of inputting equations that provides them with instant feedback. The goal is to design a framework that supports segmentation of a handwritten equation into individual symbols and returns a solution to the equation. The stroke data and bitmap image, gathered from the application, should be segmented into single characters. This data is then sent to a database for future updates to the Optical Character Recognition (OCR) engine's knowledge base. In addition, the segmented data should be sent to an interchangeable OCR black box to be analyzed. The results of the OCR engine will be combined into a single term, evaluated on the host machine, and printed to the screen of the application.

Functional Requirements

Segmentation and Analysis

- Accept user's handwritten input
  - Support digits, mathematical symbols and simple fractions
  - Single expression written on a single line only
- Split the input into bitmaps encompassing each component (a digit, an operator, etc) of the expression
- Send the segmented data to an OCR black box in the correct order
- Accept the values returned by the OCR
- Evaluate the expression and return a result

Storage

- Bitmaps for each expression are component time-stamped and stored in S3 storage
- Store the filename of the bitmap and corresponding stroke data in an Elastic Compute Cloud (EC2) database

User Interface

- Display a screen with the following:
  - Drawing canvas
  - Send button
  - Clear button
  - System status
    - Display current state (eg., “Sending data...”)
    - Display values returned by OCR and result of the expression

Stretch Goals

- Segmentation of complex fractions
- Splash screen

Non-Functional Requirements

Tools

- C++ for handwriting analysis in a Java/Objective C wrapper, allowing it to run on Android or iOS
- VM development using Vagrant and Chef
- Supported platforms: Android, iOS, and Web
- Using GitHub for source control
- Ruby/Sinatra routing to AWS platform for database storage and recall.
- AWS Server
- MongoDB for storing JSON blobs
- Node.js for front-end web application
- Code must work with the existing application (Woot Math)

Performance

- Must segment equations into three categories: digits, symbols, and fractions
- OCR must interpret segments but doesn’t need to be accurate
- The equation analysis must perfectly evaluate the OCR output
- All of the above should happen without noticeable delay

Technical Design

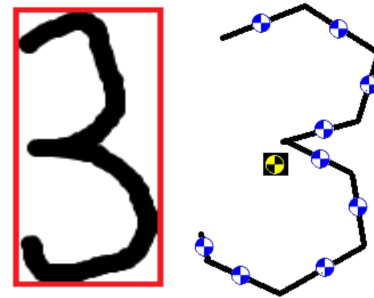
Segmentation

Segmentation is the process of taking an image and splitting it into an ordered array of sub-images. Each sub-image represents the atomic symbols that can be recognized by the Optical Character Recognition (OCR) engine. Segmentation can be placed into two categories: online and offline. Online data utilizes the time and path of the strokes, while offline data looks at a static image. These methods can be used together or individually. Though we had access to both, we chose to use offline data.

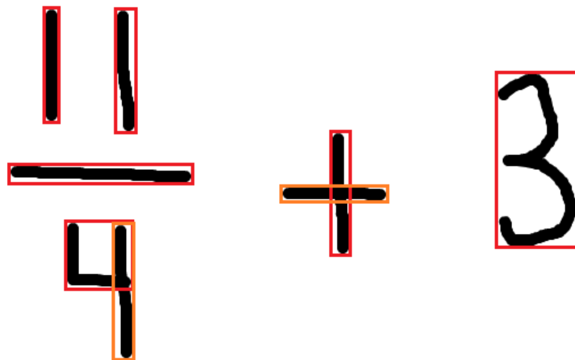
Our segmentation algorithm uses the bitmap supplied from the client as well as the stroke data from pen movement. The stroke data is a 2D array. Each sub-array consists of point data in the format [$x_location_0$, $y_location_0$, $time_0$, …, $x_location_n$, $y_location_n$, $time_n$]. Though the strokes may look continuous on the client, each point in stroke is actually the vertex of a linear polygon. A line is drawn between the x/y coordinates of adjacent points in the stroke array.

In our first attempt at creating a segmentation algorithm, we looked at several papers and tried to implement a few simple algorithms. We first categorized symbols into three groups:

- Multi-stroke Symbols: +, =, x, ÷
- Simple Fractions: ½ , ¼ , 12/13
- Simple Symbols: 1, 2, 3, /, -, (,), .

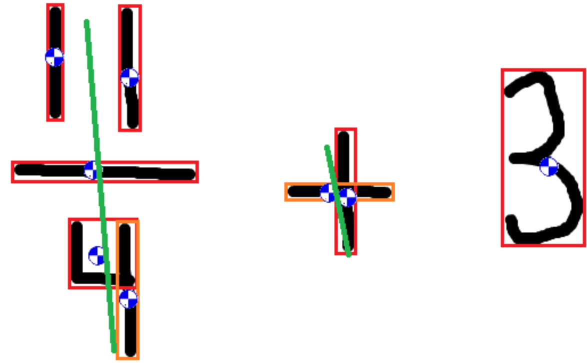


For all three groups we used the online data for quick analysis. We started by calculating the bounding box and centroid for each stroke. The bounding box for a stroke was determined by the minimum and maximum x/y coordinates in the stroke. The centroid was calculated by calculating the centroid of the lines, drawn between each adjacent point, and taking a weighted average. As a result, the centroid of a stroke was not the centroid of its bounding box and was not affected by any other stroke.



The algorithm first checks for complex symbols first, then moves to simple symbols. The initial check is for multi-stroke symbols. To find these symbols, we looked for overlaps between bounding boxes.

If they cross, we combine the strokes into a single stroke array and find its new bounding box. If the character is not a multi-stoke symbol, we check to see if a fraction is formed. We determined that there is a simple fraction if the centroids of two or more strokes create a best-fit line with an absolute angle greater than 60̊ . Once a fraction is determined, the strokes comprising it are joined into a single stroke. All remaining strokes are considered atomic and sent to the OCR.



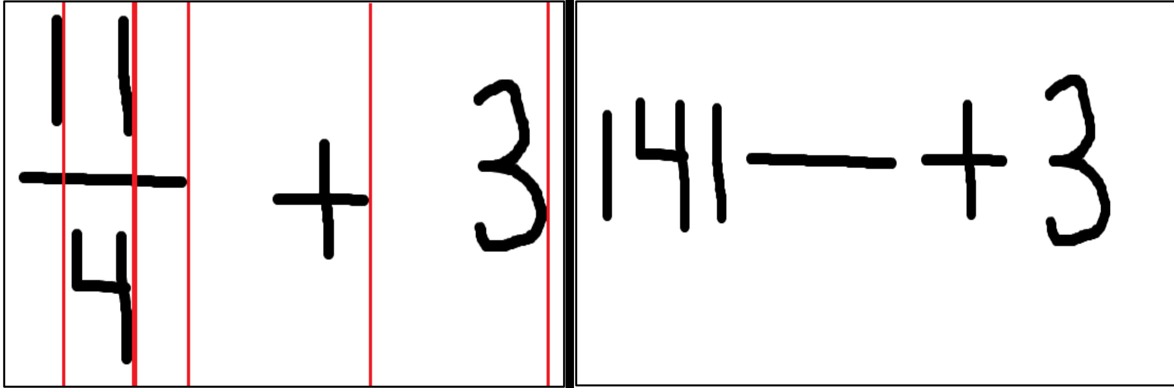
We had specific criteria to meet for the segmentation portion of this project. Our scope is limited and thus does not cover all cases. Here is a short list of what the segmentation algorithm was and was not required to do:

Required	**Not Required**
Segment digits [0-9]	Complex fractions [$\frac{2+3}{4}$ …]
Segment simple fractions [½ …]	Overlapping symbols
Segment symbols [+,-,x,=]	Multiple equations

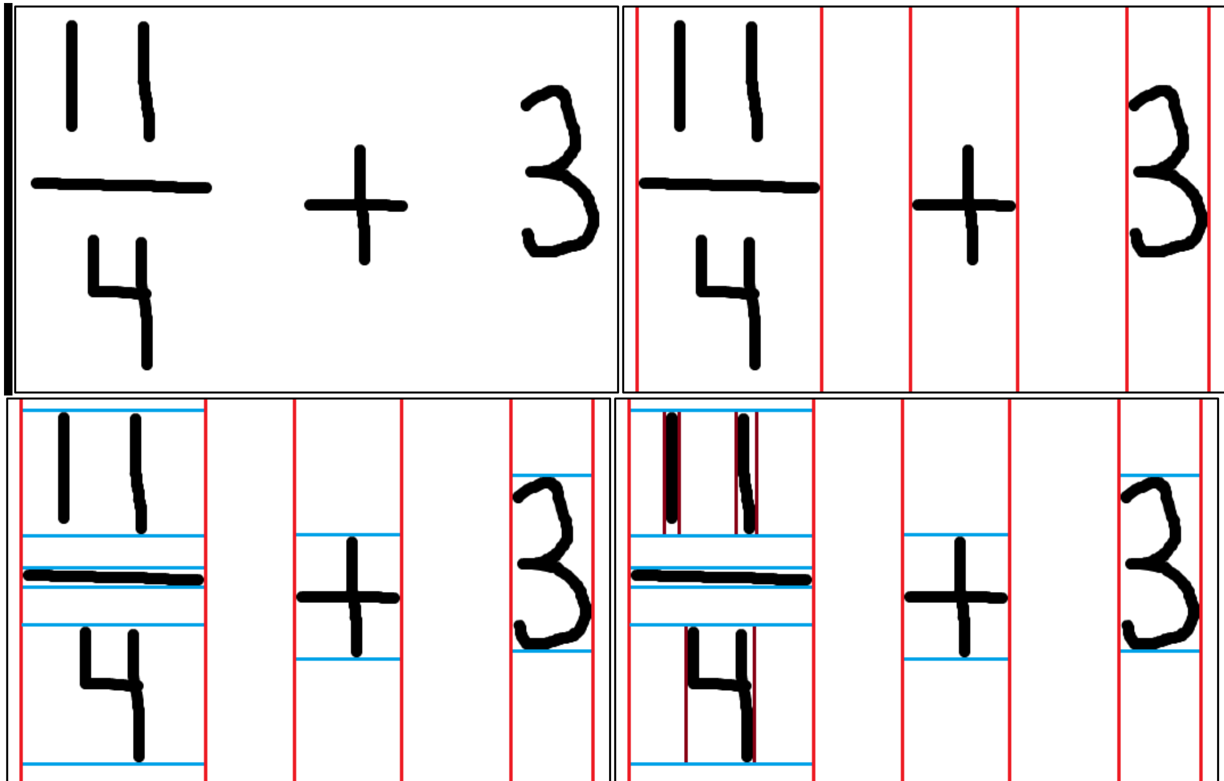
The algorithm described above was able to meet all of the required criteria above but had several major flaws. Since the stroke data was based on time, if a user wanted to go back and change something, it would completely mess up the algorithm. The bounding boxes made the writer's freedom very limited. Not only could the physical lines not cross but the boxes drawn around them could not cross either. In addition, the algorithm was not extendable. Complex fractions could not be achieved with this method and sometimes simple fractions, such as $\frac{11}{114}$, would fail because the centroids did not form a well-defined line. Even if the algorithm were lucky enough to get a complex fraction, it would not be possible to determine a fraction over a fraction. The final nail in the coffin was the fact that we were sending the OCR a simple faction as an atomic symbol, instead of the symbols comprising the fraction. This was a major flaw because the OCR could not interpret fractions and if it could, the internal segmentation in the OCR engine would make our pre-segmentation pointless.

It was time for a new approach. Another method we considered was to make an outline of each symbol. This method was rather complex and would have required a significant increase in time to process. The outlining algorithm works by analyzing each column of the bitmap. When a string of colored pixels is found it compares it to the previous column's pixels. If they are adjacent, they are part of the same symbol. Should there only be white space following colored pixels, the symbol is complete. Each symbol, in this case, is made up of a 2D array of pixels. The pixels were then reconstructed onto a blank bitmap and sent to the OCR.

While the outline of each symbol removed the bounding box issue and made OCR interpretation more accurate, it presented many more problems. Due to the horizontal movement of the algorithm, the correct ordering of the symbols was lost if a fraction or complex fraction was present. Also, complex symbols that did not overlap, such as '=' and '÷', could not be combined into a single symbol.



Our final solution addressed many of the problems and was even able to hit some of the stretch goals. In our final implementation, we took the best of both worlds. The gap algorithm we used is an offline recursive algorithm. It starts by scanning the bitmap horizontally. Each column is inspected for a colored pixel. Should white space exist between column(s) of non-white pixels, then there must be at least one symbol there. This repeats to the end of the bitmap. An array of sub-images, cropped from the bitmap at changes in white space, is created. The scanning algorithm is then called on each sub-image. This time the scanning occurs vertically across rows. This repeats, flipping rows/cols, until only a single sub-image is cropped in the vertical scan.



The result is an ordered set of atomic symbols. Though the user is still restricted due to the bounding box effect, we have opened the segmentation up to extension. Since the bitmap is static, the user is not bound by time and may freely go back and edit the equation. Additionally, the vertical scan gave us additional knowledge about the symbols being written.

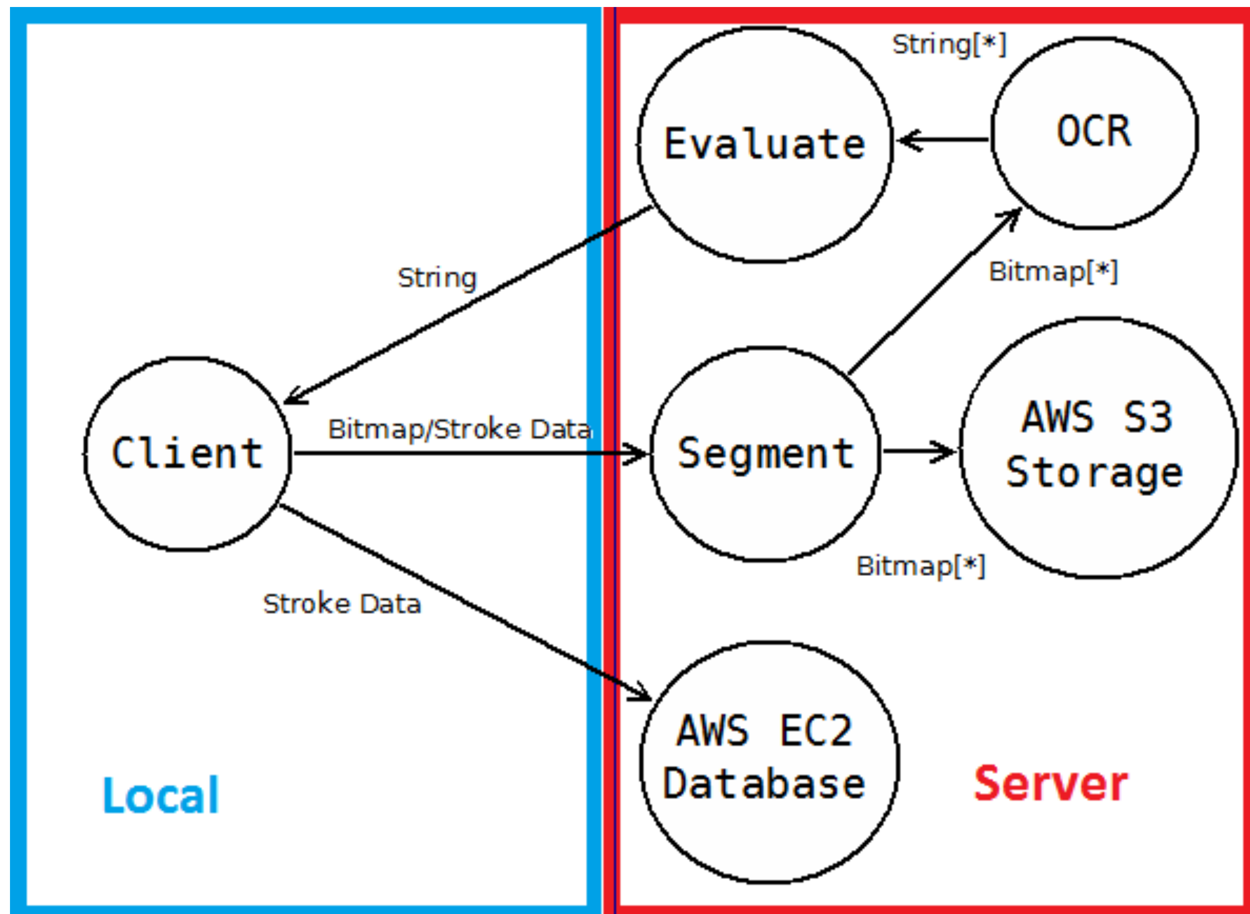
We were able to determine if a multi-stroke symbol was a fraction or '=' sign. By knowing a fraction was written, we could anticipate it for evaluation. Thus, we could send the OCR the numerator and denominator and let the evaluation take care of the division symbol. As an added bonus, the recursive nature of this algorithm let us create very complex fractions like the following:

$$\frac{\frac{\frac{11}{14}+3}{\frac{1}{12}\times 7}+4}{\frac{3}{8}-9}$$

Conclusion?

Overall, this segmentation is the best approach to the problem and gave us the best results. The down side to this algorithm is the lack of freedom on the user's part. However, the code for using online data and outlining is still in the application. This algorithm can ultimately be extended and improved with the addition of these other methods. By accomplishing the assigned goals, achieving some of the stretch goals, and allowing further stretch goals to be made; we believe that we have met the criteria for a good algorithm.

System Architecture



Design

Specifications

The problem associated with implementing such technology is the need for a consistent, accurate, and efficient OCR engine that ideally executes as the student writes. Regardless of whether offline or online recognition is used, the program needs to capture input information, accurately interpret it, and evaluate the expression if possible. Handwritten data coming from the client-side digit recognition applications is stored in straight blobs, which includes bitmap data and stroke data. The OCR engine then analyzes the data locally and returns an interpreted expression.

The program will also need to send packets of the information it receives. Using Amazon Web Services (AWS), the collected stroke data will be stored in EC2 and the bitmap will be stored in an S3 storage system. The database provides information to the handwriting recognition software, tailoring its recognition of individual users' handwriting. This data will be retrieved when the knowledge base for the OCR algorithm is being updated.

Usage

This addition to the program is important for both teachers and students. It will allow students to work intuitively when solving problems. Since the text will be uniform once converted, a teacher viewing the student's progress can easily and accurately follow the student's work flow. If a teacher creates a hint or solution to the problem, students can have a better understanding of the material covered. Finally, the training data gathered from user input can be used to train the recognition algorithm and further advance the recognition of any user's writing, improving and broadening functionality of the program over time.

Benefits

The top, key benefits of the design are:

- Reception and interpretation of user input
- Efficient feedback and expression evaluation for the student
- Training data from user input training the recognition algorithm

Assumptions

All assumptions of this design include:

- Functional and efficient OCR library
- Efficient conversation between the client and local storage
- Efficient conversation between the client and AWS

Risks

Programming risks:

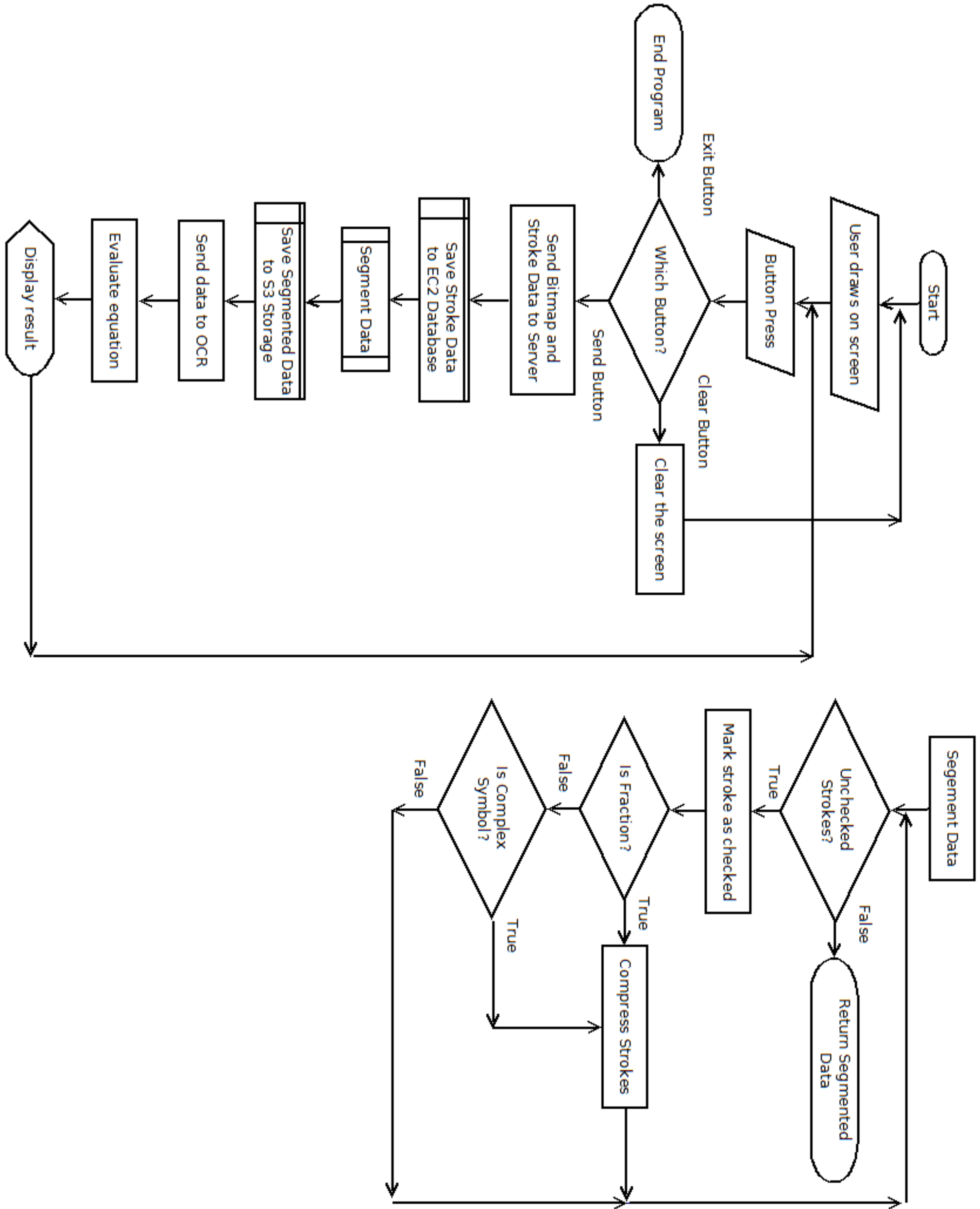
- Varying levels of experience with necessary languages
  - Ruby
  - Sinatra
  - ActionScript 3
- Varying levels of experience with necessary applications/services

        - MongoDB
        - AWS
        - Kudu
    - Multiple projects to consider (input recognition and data collection/analysis involve different processes and different requirements). Resulted in initial disorganization and stress.
    - Miscommunication with the client. Resulted in delayed clarification of project requirements, specifically whether the scope of the project included implementation of an OCR engine.

Server and application risks:

- Dramatic failure in segmentation of the expression could result in poor functionality and a poor user experience.
- Possibility of database failure or related connectivity issues. As a result, the OCR would not be able to update its knowledge base and its performance could suffer.
- Although preliminary implementation uses fake S3 and EC2 servers, the final product will be vulnerable to outages in the AWS service. If the OCR engine remains on the server side, core functionality of the app will be significantly reduced.

Use Case Diagram



Design Decisions

We chose to use *Google Tesseract* as our functional OCR, because even though *Google Tesseract* is not the ideal OCR (Optical Character Recognition) to use with handwritten data, we are using a command-line implementation of it on a Virtual Machine in order to simulate the process of sending handwriting data to a handwriting-recognition system--the real recognition system that will be used in the end system is a black box which will later be implemented by Nimbee themselves, but *Tesseract* will suit our purposes for implementing and testing the rest of the program.

We have chosen to use a Sinatra file (written in Ruby) in order to communicate collected data to storage and database systems. This is done because Sinatra works well in regulating data traffic and Ruby is the only language which uses it, Ruby itself affording us a powerful and readable language that allows us to perform a number of different functions. We will also use Ruby because it allows us to use a powerful eval() function with which we can evaluate math expressions and strings.

We have chosen an Amazon Web Services S3 (Simple Storage Solutions) instance for storing bitmap data. It is a fake one, for our developing purposes, as S3 storage is not free. This is chosen because it's a simple binary blob storage system, and because the client wants it for its wide distribution. The bitmap data is stored in a simple S3 bucket because it can be used for training the OCR.

We have chosen a local MongoDB instance for storing a JSON blob, collecting stroke data consisting of location and time points. Nimbee intends to replace the local MongoDB instance with an Amazon Web Services EC2 (Elastic Compute Cloud) using MongoDB. The reason for this is that Amazon services are highly scalable and secure. MongoDB is also highly scalable and is suited for storing JSON objects.

We have chosen to implement the client program in ActionScript 3. The bulk of what the user sees is written in ActionScript 3 because it is what was suggested by the client, Nimbee, who in fact provided a collection of boilerplate code in which we could begin application development. AS3 allows us to develop for all three of the client's target platforms: iOS, Android and Web.

The ActionScript 3 application also uses the Starling and Feathers user interface control frameworks in order make the product more palatable and interesting for the end user--we chose these frameworks because they are simple to use and made for developing slick programs with simple user interfaces (for example, it is used with Angry Birds, etc.). These were implemented by way of using higher resolution button graphics and similar ideas.

Results

The project goal was to implement a framework that allows a handwritten function to be analyzed. The data collected from the handwriting should be used to segment and recognize the components of the function, which should then be evaluated and returned to the client side. Additionally, the framework must store the handwriting data for later use in training the recognition algorithm. The framework we implemented succeeds in all of these tasks.

The project's primary constraint was the lack of an Optical Character Recognition (OCR) algorithm suited for identifying handwritten text. The current OCR being used is Tesseract, an engine designed to recognize printed, typed text. Due to the subpar performance of Tesseract, testing the segmentation of the characters was done by manually opening the files created by the segmentation process. If a more suitable OCR had been available, the segmentation testing could have been done through inspection of the values returned by the OCR engine.

This project has taught us several valuable lessons, especially concerning communication with the client. Initially, the project's specifications were not well-defined and the team's progress suffered as a result. These issues showed us the importance of clear and direct communication with a client, especially during the early phases of a project, when the goals are more nebulous. We also came to understand the distinction between healthy struggles versus unproductive struggles while coding. The client used a proprietary service that we initially did not understand. Rather than seeking help with the service immediately, we attempted to solve some of the issues on our own. This time would likely have been better spent tackling a problem within the scope of the project, rather than trying to solving problems whose solutions were just an email away.