UROPS 19/20 Report
Optical Character Recognition for Math Equations
Project Code 1920-00054

Gao WenHan A0172770A
Supervisor: Vikneswaran S/O Gopal Senior Lecturer

April 25, 2020

Abstract

This is a project which attempts to develop a pipeline for detecting handwritten mathematics equations, and then assessing their correctness through automatic grading. The foundation of this project is the *Seshat* parser developed by Francisco Álvaro [1]. We also make use of properitory softwares and libraries such as *Xournal++* and *Sympy* in the construction of the pipeline architect. In this work, we mainly focus on mathematical expression equality and also solving for unknown variable.

Contents

1 Introduction and Motivation

In most mathematics classes today, a teacher is responsible for marking many similar scripts. This manual process can be very repetitive. As artificial intelligence continous to develop, we seek to explore a way that can automate this process. Mathematics questions can be broadly separated into two categories. The first is expression equality where one needs to show expression on the lefthand side is equal to the righthand side (1), using a series of equal expressions which links both together.

$$x^2 + 2x = (x+1)^2 - 1 \tag{1}$$
$$x^2 + 2x = x^2 + 2x + 1 - 1 \tag{2}$$
$$= (x+1)^2 - 1 \tag{3}$$

The other category is where an equation with unknown variable(s) (4) is given and one needs to solve for the unknown.

$$2x + 1 = 3 \tag{4}$$
$$x = 1 \tag{5}$$

In this project, we will be focusing on these two categories of mathematics questions and propose an end-to-end data processing pipeline that will perform automated grading for them.

2 Literature Review

2.1 SCG Ink

The SCG ink format is a text-based representation of the (x, y) coordinates of the strokes drawn by the user [2]. Each SCG ink file contains the total number of strokes appearing. For each stroke, there is a number indicating the number of points appearing in that stroke and all the (x, y) coordinates of those points. Therefore, it is able to capture the sequential relationships of the input strokes as compare to the conventional pixel image. The parser takes a SCG ink file as input and makes use of this sequential infomation of the strokes to determine the most likely mathematical symbols.

2.2 Seshat Model

2.2.1 Statistical Framework

The paper [1] propose modelling the structural relationship of a mathematical expression using a statistical grammatical model. The authors define the problem as obtaining the most likely parse tree given a sequence of strokes of mathematics input. The tree derivation will produce the sequence of pre-terminals that represents the structure that relates all the mathematical symbols. In order to generate the sequence of pre-terminals, the parser will take into account all the strokes combinations to form possible symbols.

Therefore, the problem can be futher divided into two. First, the segmentation and recognition of symbols. Second, the structural analysis of the maths expression. We can see that these two problems are closely related and are interdependent. Hence, the proposed strategy computes the most likely parse tree while smultaneously solving for symbol segmentation, recognition and structural analysis of the input. More formally, the problem is defined as such

$$\hat{t} \approx \underset{t \in \mathcal{T}}{\operatorname{argmax}} \max_{\mathbf{s} \in \mathcal{S}; \mathbf{s} = yield(t)} p(\mathbf{s}|\mathbf{o}) \cdot p(t|\mathbf{s})$$

where $\mathbf{o}$ is the sequence of input strokes, $\mathbf{s}$ is the sequence of mathematical symbols derived from the strokes, and $\hat{t}$ is the most likely parse tree. This defnition assumes that the structural part of equation only depends on the sequence of the pre-terminal $\mathbf{s}$. $p(\mathbf{s}|\mathbf{o})$ represents the symbol likelihood and $p(t|\mathbf{s})$ represents the structural probability.

2.2.2 Symbol Likelihood

A symbol is made up of one or more strokes. In the paper, the authors define a symbol likelihood model that is not based on time of the information but rather the spatial information as symbols can be made up using non-consecutive strokes. However, it is reasonable to believe that only strokes that are close to one another will form a mathematical symbol and this will greatly reduce the search space for the possible symbol segmentations. Overall, the authors futher factor the symbol likelihood into three models: a symbol segmentation model, a symbol classification model and a symbol duration model.

The symbol segmentation model uses the concepts of visibility and closeness of the strokes to generate a graph G with each stroke as a node and the edges only connect strokes that are visible and close. Then a segmentation hypothesis is valid only if the strokes it contains form a connected subgraph in G. This is able to reduce the set of possible strokes segmentation. Then for each stroke in a possible segmentation, geometric features are generated, such as mean horizontal position, mean veritical position etc. All these features are trained using a Gaussian Mixture Model (GMM).

The symbol classification model uses two Bidirectional Long Short-Term Memory Recurrent Neural Network (BLSTM-RNN) classifiers. RNNs remembers the past and it's decisions are influenced by what it has learnt from the past [4]. Unlike a normal Neural Network where outputs are influenced by the weights applied to the inputs, there is also a hidden state vector in RNN that represents the context based on prior inputs or outputs. Hence, the same input can produce a different output due to diference in the previous inputs in the series. In handwriting recognition tasks where there could be considerable ambiguity given just one

part of the input, a bidirectional RNN allows the network to access the context information in both time directions and detect the present. Lastly, Long Short-Term Memory is an advance architecture that allows cells to access context information over long periods of time.

Two classifier models are separated into online and offline. The online BLSTM-RNN uses online features of the points such as normalized (x, y) coordinates, normalized first and second derivatives etc. At the same time, the a binary image representation of the input is rendered and processed. Then several offline features of the image such as number of black pixels in the column, center of gravity of the column etc. The probabilities computed by these two classifiers are then combined using linear interpolation and a weight parameter.

Finally, the symbol duration model accounts for the intuitive idea that a mathematical symbol class is usually made up of a fixed number of strokes. Hence, a simple way to calculate the probability of of observing l_i strokes given a certain symbol class is proportion of the number of times that symbol class was composed using l_i strokes over the total number of that symbol class in the set.

2.2.3 Structural probability

The authors define a generative model based on a two-dimensional extension of the well-known context-free grammatical models to compute the parse tree probability. Context-free model is able to represent the structure of natural language and hence can be extented in to mathematics notations. This is because there are also structural dependencies between different elements in the mathematics expression. The parse tree probability can be factor into the probability of the rules of the gramma and the probability of two region that can be arranged according to a spatial relationship r. A spartial relationship model is used to model the later probability. Given two regions, the authors defined nine geometric features and then trained a GMM to compute the posterior probability for each spatial relationship r given any two regions. This posterior probability is then offset by a penalty function based on the minimum distance between the two regions.

2.2.4 Parsing Algorithm

The parsing algorithm is a dynamic programming method. At the initialization step, the probability of every valid segmentation is computed. At the general step, the parsing algorithm computes a new hypothesis by merging perviously computed hypotheses until all strokes are parsed. Then the most likely hypothesis and its associated derivation tree $\hat{t}$ is retrieved.

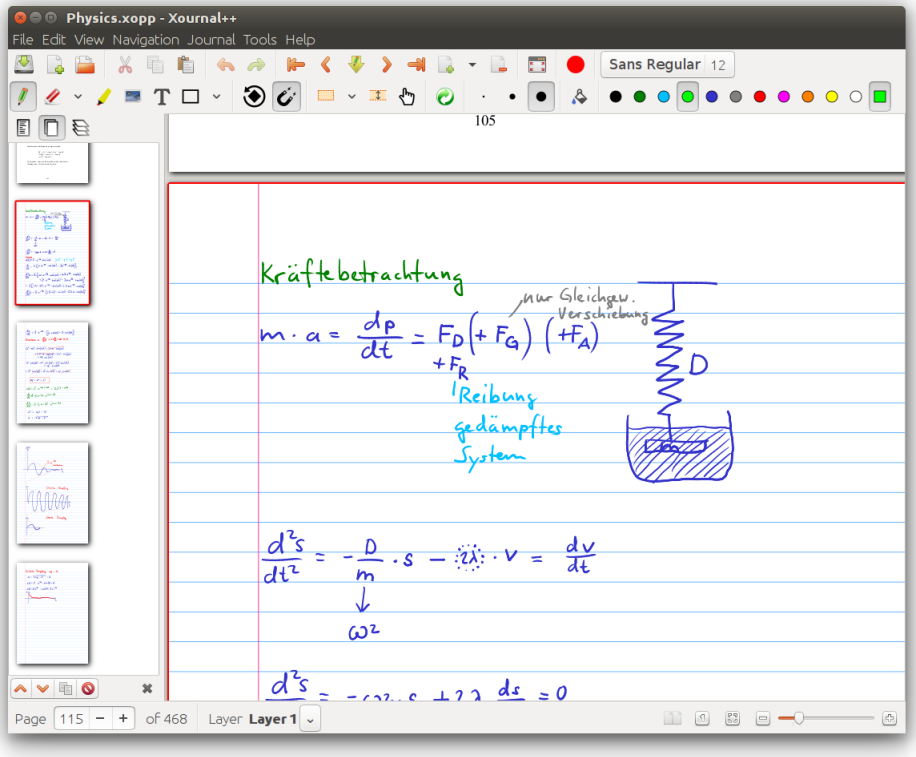
3 Pipeline Architecture

In this section, we will be describing the entire data pipeline, from raw data input to final grading outcome. The pipline is divided into three main parts: input, processing and transforming, and finally correctness evaluation. The entire process, except for data input, will be done using python and some common libraries such as *beautifulsoup* and *sympy*.

3.1 Data Input

The *seshat* parser requires data with sequential information. This means that we cannot simply scan documents as it does not capture the vital information needed. Hence, we have explore a few alternative methods of digital pen input which are able to capture the point-wise coordinates of the strokes. The open-source software *Xournal++* availiable on Github is a good fit for our needs. It is a handwriting notetaking software with PDF annotations support. It can be run on Windows 10, macOS and Linux. More specifically, it supports digital pen input from devices such as Wacom Tablets.

Figure 1: A screenshot of Xournal++ from Github



The file format for the software is *.xopp. It is actually an XML that is .gz compressed. A python script is used to unzip the .xopp file into XML file by using python libraries such as *gzip* and *lxml*. The resulting XML file is a tree structure representation with different branches for every page of the document. Within each page branch, different sub-branches are used to contain each stroke and its points' coordinates.

3.2 Data Processing

As a full mathematic proof or solution can be written in many different ways and structures on a piece of paper, it becomes too complex for the parser to take into account all possible scenarios. Hence, we have made a few assumptions that we believe are intuitively common

so that the problem becomes simpler. Below are the assumptions we have made with respect to the hand writting format and structure.

1. All mathematical expressions are written line-by-line sequentially starting from the first line.

2. For proving expressions equality, only one expression is written per line. For example

$$\begin{aligned} & x^2 + 2x \\ = & x^2 + 2x + 1 - 1 \\ = & (x+1)^2 - 1 \end{aligned}$$

   However, this assumption presents some challenges with regards to the accuracy of the parser which we will be elaborating later.

3. For solving mathematical equations, only one equal sign will be observed for each line. For example

$$\begin{aligned} 2x + 1 &= 3 \\ x &= 1 \end{aligned}$$

4. Any proof or solution to a question is only confined on one page. Every new question or sub question must be started on a fresh page.

We also derive a way to represent and store the ncessary data using python object-oriented programming. We have created a Page class and a Document class which aim to reflect the information structure in real life closely.

3.2.1 Page Class

As mentioned previously, we have made the one-question-per-page assumption. Hence, one Page class object will reflect the solutions for one question. The attributes and methods of a Page class is quite intuitive as follows

1. lines: a (key, value) dictionary object with key indicating the line index (number) and value as the parsed *Latex* string for the corresponding lines in the document

2. numlines: an integer indicating the number of lines of mathematical symbols on the page

3. getNumLines(): return the number of lines of the page

4. printLines(): print all the lines on the page

5. getLines(n): return the *nth* line of the page as a *Latex* format string

6. evaluate(type): *type* can be either *solve* for equation solving or *reduce* for proving expression equality. It returns *Correct* if the solution follows a logical sequence. If the logical sequence breaks down in the *nth* intermediate step, it will return *Fail at line n*. The logical evaluation is done using the *sympy* library which we will be elaborating later in section 3.3

One of the limitations of the *seshat* parser is that it is much more accurate when parsing single-line mathematics symbols than multi-line full page. Hence, we propose a way to divide the input data of the SCG INK file into smaller files containing data of only one line each. These files will then be parsed sequentially and loaded into the Page class object. Identifiying lines in a page presents its own set of challenges. However, we are able to leverage onto the user interface of *Xournal++* to simplify the process. As seen in Figure 1, the template background of *Xournal++* closely resembles a typical paper worksheet with gridlines. We assume that users will base their hand writting on the gridlines and hence, the problem reduces to finding the coordinates of the gridlines.

While we cannot simplify extract the coordinates from the template as the gridlines are only a visual aid, we have manually drawn lines overlaying each of the gridlines. Then, we are able to extract out the (x, y) coordinates of each gridline and save them as a *pandas* data frame which is exported as a pickle file.

The pseudo code for initializing the Page class object is as follows:

1. Load data frame of (x, y) coordinates of the gridlines as *template*
2. Load full data from orginal SCG INK file
3. For each stroke,
   (a) Collect all (x, y) coordinates into a list separately
   (b) Calculate the mid-point using $\frac{1}{2}(\max\{ycoordinate\} - \min\{ycoodinate\})$
   (c) Iterate through the rows of *template* until mid-point is between the y-coordinates of two consecutive lines. Label the stroke with that particular line index
4. For each line that contains strokes, create a new SCG INK file using the x and y coordinates lists
5. Parse the new SCG INK files using *seshat* and save the outcome *Latex* string into the *lines* attribute

3.2.2 Document Class

Intuitively, the Document class is a representation of the entire work document. It can be viewed as a collection of Page class, one for each question. The input is the raw xopp file from *Xournal++*. The class object has the following attributes and methods:

1. npages: the total number of pages in the document

2. pages: a (key, value) dictionary object where key is the page index and the corresponding value is a Page class object

3. getPage(n): returns the Page class object at index n

4. numPage(): returns the total number of pages

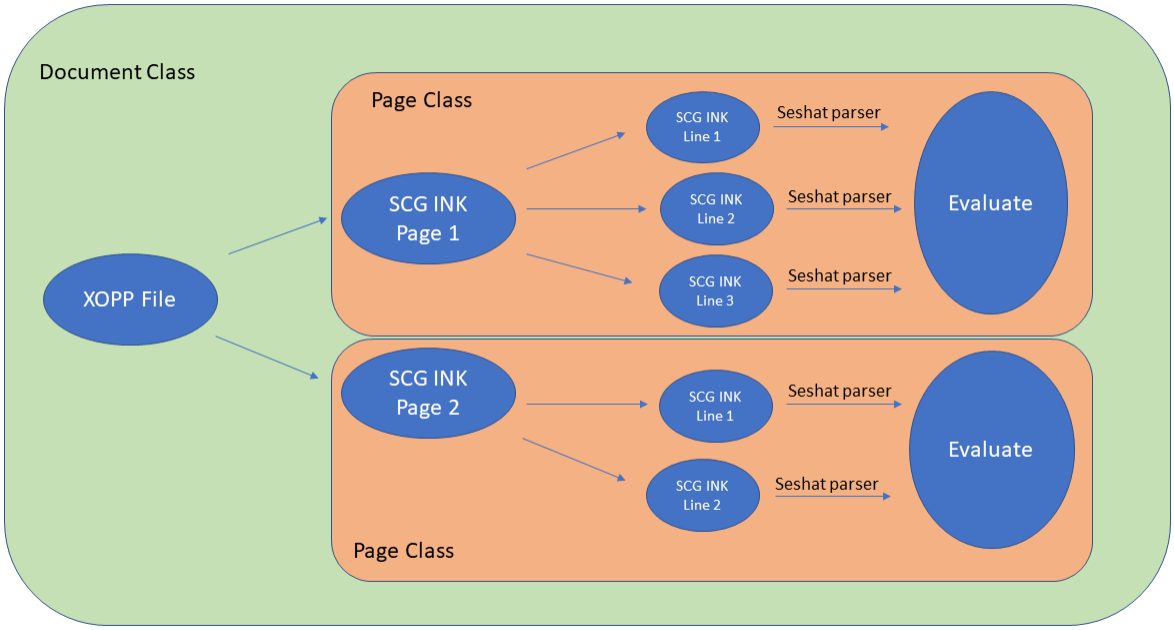
5. toLatex(): generate a *Latex* document

6. evaluatePage(n): evaluate the Page class object at index n

7. prettyPrint(): print the content of the Document class in a readable format

Since *seshat* is very restrictive in the input file format, data processing is require to transform the xopp file to the acceptable SCG INK file. As mentioned before, we first unzip the xopp file into a XML file. We are only interested in the coordinate information which are contained under the <*stroke*> tags. There are many ways to extract information based on tags and we will use the most commonly used library which is *BeautifulSoup*. The library allows us to first find all <*page*> tags so that we are able to split the data by pages. Within each page, we look for all the <*stroke*> tags which contain the coordinates information. We then simply store these coordinate into lists and generate a SCG INK file for each page using the lists of coordinates. These new SCG INK files will be use to initiate the individual Page class objects. Using an example of a document of 2 pages with 3 and 2 lines each, Figure 2 shows an illustratuon of the overall data structure and process flow.

Figure 2: Data Pipeline



3.3 Data Output

As stated previously, the evaluation of correctness is done using the *sympy* [3] library. It is an open source tool that is commonly used when dealing with symbolic mathematics. It has many features such as basic arithmetic, expansion of polynomials, differetiations, integration etc. For our basic usage, we will be focusing on simplification of polynomials and solving algebraic equations. One important function from the package we will be using is *simplify()*. This function attempts to apply all the functions in *sympy* which perform various kinds of simplification to the input expression to arrive at the simplest form possible. Another function we will be using is the implication expression >>. For example, a logical expression of a implies b is represented by $a >> b$. This expression can then be fed into the *simplify()* function which will give a logical output.

3.3.1 Evaluating Expressions Equality

For two expressions to be equal, their difference must be 0. Using this fact and the assumption (2) made in section 3.2, we derive the following iterative algorithm:

1. Let p_{old} be the first expression on the first line
2. For $i = 2, .., n$ where n is the last line
   (a) Let p_{next} be the expression on the ith line
   (b) Let r = simplify($p_{next} - p_{old}$).
   (c) If n is the last line and r == 0, return *Correct*. Else if r == 0, let $p_{old} = p_{next}$. Else, return *Fail at line i*

In essence, the algorithm repeatedly compare the current and previous expression by taking their difference. Whenever the difference is not zero, the loop breaks and informs the user the step which has failed the test.

3.3.2 Solving Equations

For two equations to be equal, we can view it as given that the first equation is correct, the same set of unknown variables must also satisfy the second equation. This means we can view the equations as alternating prior and posterior conditions that the unknown variables have to satisfy. Using this framework and the assumption (3) made in section 3.2, we derive the following iterative algorithm:

1. Let p_{old} be the first expression on the first line
2. For $i = 2, .., n$ where n is the last line
   (a) Let p_{next} be the expression on the ith line
   (b) Let r = simplify($p_{old} >> p_{next}$).
   (c) If n is the last line and r == True, return *Correct*. Else if r == True, let $p_{old} = p_{next}$. Else, return *Fail at line i*

Similar to the previous case, the algorithm repeatedly evaulate the correctness of the current equation by using the previous equation as a prior condition. Whenever the equation fails, the loop breaks and informs the user the step which has failed the test.

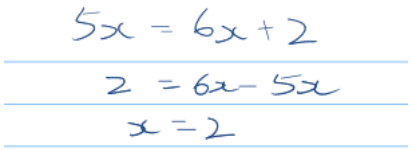
4 Reviews

In the last section, we describe the difficulties and limitations faced when testing the data pipeline on samples. We also propose some future extension to the project and possible alternative that can be considered.

4.1 Limitations

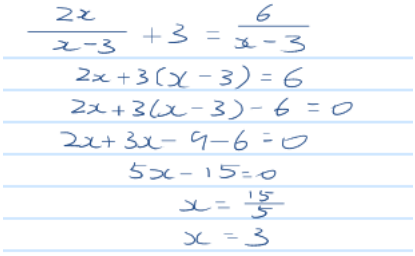
The effectiveness of the the entire process mostly hinges on 2 factors. Firstly, the accuracy of the *seshat* parser is critical. It is crucial that the parser is able to recognize and output the correct *Latex* string that represents the input handwritting.

Figure 3: Example 1



Given a simply polynomial equation as shown in Figure 3, the parser has no problem recognising the respective symbols. It is easy to see that there is an error in line 2 where 2 on the left-hand side should be -2 and *sympy* is able to detect the logical flaw correctly. However, when the equation becomes more complicated as shown in Figure 4, the parser has a difficult time recognizing all the symbols correctly.

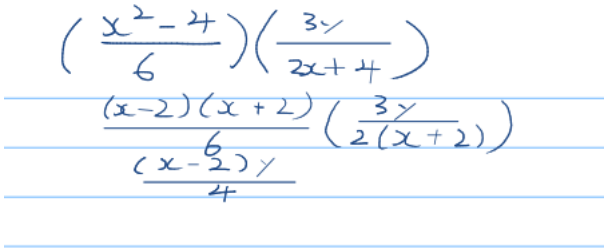
Figure 4: Example 2



Since the geometric features are used in the predictive models, we suspect that the handwrittings that are narrow and close to one another can be a challenge to the parser. This may explain the increase in error rate when the equations or expressions become more complicated and dense together.

Another challenge occurs when dealing with comparing expressions. Base on our input format, expressions are written one per line. The normal convention is to add a "=" sign at the beginning of every subsequent line. However, the parser has a hard time detecting and recognising the equal sign at the beginning of the expression correctly. As a result, an abitrary input restriction is enforced where equal sign is made unnecessary as seen in Figure 5.

Figure 5: Example of Expression Input



We have also tested the parser to detect differentiation and integration handwritting. For an integration example as shown in Figure 6, the parser is able to correctly recognise the integral sign as well as the superscript and subscript on line 2. However, the problem arises from the usage of *sympy*. The representation norm on line 2 does not comply to the *sympy* input format and cannot be directly feed into the *simplify()* function. As a result, our existing proposed algorithm is not able to evaluate it.

Figure 6: Example of Integration

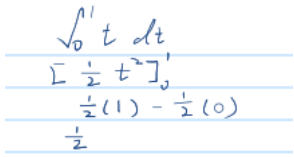
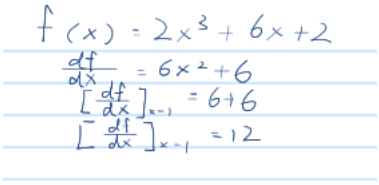


Figure 7: Example of Differentiation



Similar problem arises when it comes to differentiation as seen in Figure 7. While the parser is still able to recognise the symbols correctly (abeit with more difficulty for $\frac{df}{dx}$), *sympy* is unable to recognise the representation style used on line 3.

The limitations we have mentioned above are not exhausive and there can be potentially more issues as we continue to test the program. In essence, the major issue that we need to overcome is the accuracy of *seshat*. As the model is already pre-trained, we do not know the extent of the training and the types of samples used. Hence, it is vital that we continue to generate more variety of samples to train the model and continue to improve its accuracy. Furthermore, the two algorithms described to evaluate the correctness of the mathematical working relies heavily on strict assumptions and fixed input structure. Clearly they are unable to perform well for more advance cases such as substitution, differentiation, integration etc. Therefore, more data processing is needed to parse the *Latex* string into usable form for the various *sympy* functions to evaluate. The proposed evaluation algorthms will need to be more flexible and generalize to be able to account for different types of situations.

4.2 Alternatives and The Future

There are other handwritting recognition softwares available that can be used instead of *seshat*. The *Math Input Panel* is a built-in program in Windows 10 which is able to convert pen input mathematical writting to string form. One advantage of it is that it is able to perform real-time recognition. The user is also able to lasso and select an area to edit if the output string is not as desired. However, challenge of this program is that the raw data output is not made available to users. Instead, it makes use of the clipboard and immediately transfer the output string to another opened programs such as *Microsoft Word*. Hence, we are unable to parse the output again using *sympy* to perform evaluation.

While the project is only at its infancy stage, we are able to demostrate the possibility of automated grading using a combination of handwritting recognition machine learning program and symbolic mathematical evaluation program. The proposed data pipeline is only able to deal with naive and simple examples. More work needs to be done to improve on the two main limitations as mentioned in section 4.1.

Having said that, the aim of the project is not to achieve full grading automation. Rather, the goal is to target questions that have a very fixed and predictable format and hence, reduce the amount of work and time spent by the human marker. This is because script marking is repeatitive especially when the number of scripts is huge. In these cases, questions can be separated and those conforming to our proposed program will be evaluated efficiently. This not only save precious human hour and cost, it also allows the markers to spend more time on more open-ended questions so that overall marking error can possibly reduce. Furthermore, the program has the possibility of improving the learning experience of students by using it as a way of self-check. It is able to provide instant feedback to students and alert them of the location of the error that they are making. This can aid in the learning process by allowing the instructor to spend more time on teaching the concept instead of spotting errors for the students.

Lastly, we can consider the different levels of education where this program can be used. In primary levels, mathematics are mostly taught using pictures and drawings. The pro-

portion of questions requiring equation and expression writing is not high which can further limit the usage of the program. As such, the program may be more useful for secondary and tertiary education where more algebraic expressions and equations are being used.

5 Conclusion

In conclusion, the proposed program is able to perform relatively well in terms of recognition and evaluation when the examples are simple enough and fall strictly within the limitations set. It aims to support the human marker rather than totally replace him. With continuous improvement, it has the potential to create a more efficient way of math script marking as well as improving the learning experience of the students.

References

[1] Francisco Álvaro, Joan-Andreu Sánchez, and José-Miguel Benedí. "An integrated grammar-based approach for mathematical expression recognition". In: *Pattern Recognition* 51 (2016), pp. 135 –147. ISSN: 0031-3203.

[2] Scott MacLean et al. "Grammar-based techniques for creating ground-truthed sketch corpora". In: *International Journal on Document Analysis and Recognition (IJDAR)* 14.1 (2011), pp. 65–74.

[3] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: (2017).

[4] Mahendran Venkatachalam. *Recurrent Neural Networks*. 2019. URL: `https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce`.