Automated Grading of Handwritten Numerical Answers

Mark Brown

mbrown73@calpoly.edu

mark@mtbrown.me

Adviser: Dr. Dennis Sun

dsun09@calpoly.edu

Computer Engineering Program

California Polytechnic State University

San Luis Obispo, CA

June 2017

Contents

1 Requirements **2**

2 Design **2**

2.1 Answer Sheet 2
2.2 Detecting the Answer Locations 4
2.2.1 Template Matching 4
2.2.2 Contour Detection 4
2.3 Digit Classification 5
2.3.1 Model Architecture 5
2.3.2 Model Training 7
2.3.3 Model Calibration 7
2.4 Grading 8
2.4.1 Direct Grading 8
2.4.2 Indirect Grading 9
2.5 Aggregate and Threshold Selection 10
2.5.1 Aggregate Selection 11
2.5.2 Threshold Selection 11
2.6 Interface 12
2.6.1 Grading Scans 12
2.6.2 Calibrating Models 13
2.6.3 Report 14

3 Results **17**

3.1 Future Improvements 17

4 Appendix **18**

4.1 Source Code 18
4.2 Senior Project Questionnaire 18

Abstract

The objective of this project was to automate the process of grading handwritten numerical answers in a classroom setting. The final program accepts a scanned answer sheet completed by the student along with a description of the correct answers and produces a detailed report describing the confidence of correctness for each answer.

Computer vision techniques are used to automatically locate the locations of the answers in the scan. Each digit is then passed through a convolutional neural network to predict what was written by the student. The individual probabilities of each digit produced by the network are aggregated into a single score describing the model's confidence in the correctness of the answer.

1 Requirements

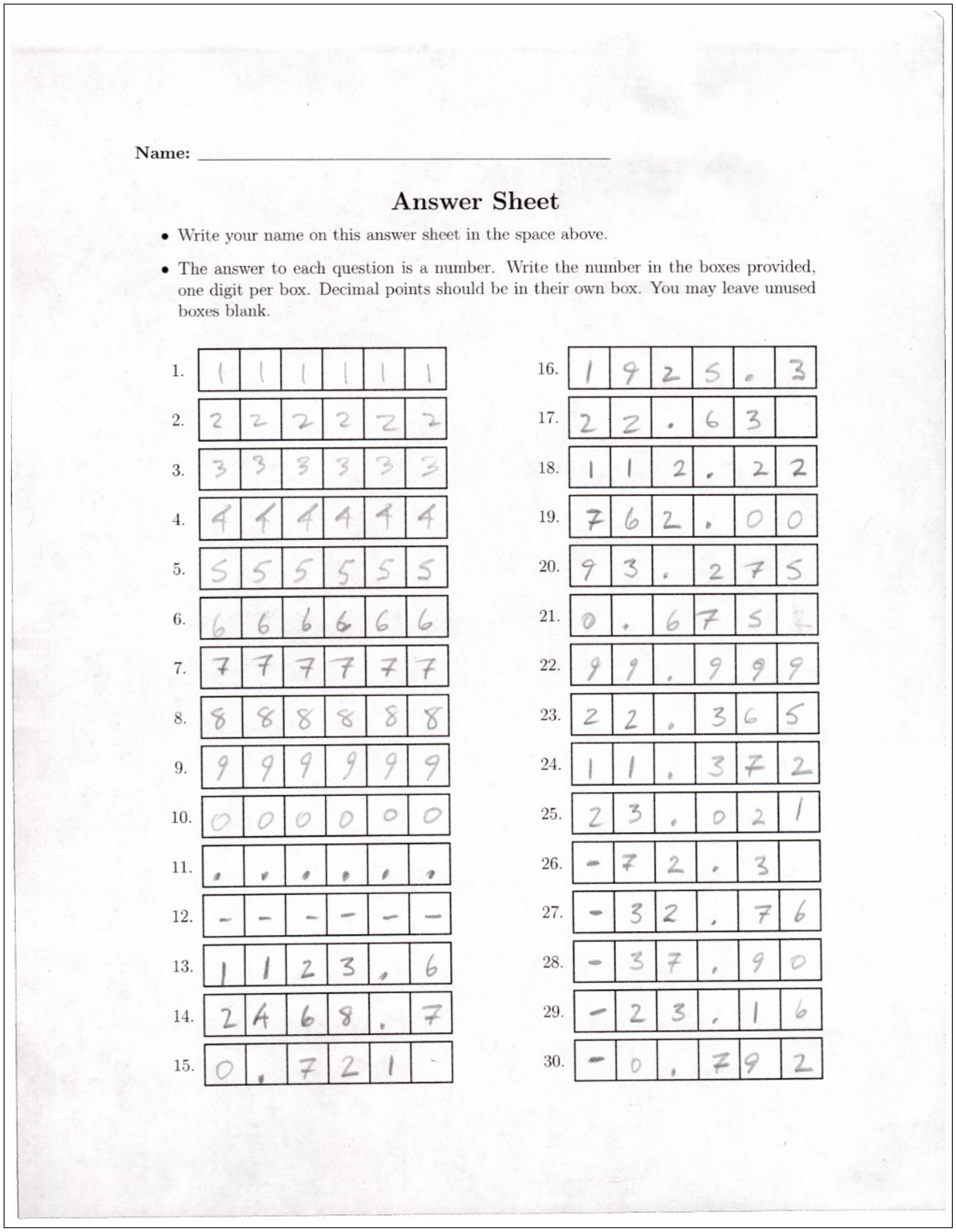
The following requirements were identified at the start of the project:

- Interface
  - The program should accept images of scanned answer sheets along with the correct answers.
  - The program should support batch grading multiple scans in a single execution.
  - The program should be usable by the average professor.
- Answer Sheet
  - The answers should be automatically located and extracted from the answer sheet.
  - The answer sheet should be simple to use from the student's perspective.
- Grading
  - The digits 0-9, decimal points, and negative signs should all be supported characters for students to write.
  - The algorithm should support variations in representation such as an optional leading 0 before a decimal point or optional trailing 0s after a decimal point.
- Output
  - The output should classify each answer as correct or incorrect and include a metric for the confidence in each answer's correctness.
  - It should be possible to determine the reasoning behind each answers score with a visual report.

2 Design

2.1 Answer Sheet

The answer sheet was designed to be straightforward for the students to use while also remaining easy to parse with software. The answer sheet consists of a series of numbered questions, each of which contain clearly defined boxes for students to write the digits of their answer within. The well-defined edges and the repetitive pattern make it easy to automatically detect and identify the boxes of each answer. A sample completed answer sheet can be seen in Figure 1.



Name: ______________________________

Answer Sheet

- Write your name on this answer sheet in the space above.
- The answer to each question is a number. Write the number in the boxes provided, one digit per box. Decimal points should be in their own box. You may leave unused boxes blank.

1.	1	1	1	1	1	1
2.	2	2	2	2	2	2
3.	3	3	3	3	3	3
4.	4	4	4	4	4	4
5.	5	5	5	5	5	5
6.	6	6	6	6	6	6
7.	7	7	7	7	7	7
8.	8	8	8	8	8	8
9.	9	9	9	9	9	9
10.	0	0	0	0	0	0
11.	.	.	.	.	.	.
12.	-	-	-	-	-	-
13.	1	1	2	3	.	6
14.	2	4	6	8	.	7
15.	0	.	7	2	1	
16.	1	9	2	5	.	3
17.	2	2	.	6	3	
18.	1	1	2	.	2	2
19.	7	6	2	.	0	0
20.	9	3	.	2	7	5
21.	0	.	6	7	5	
22.	9	9	.	9	9	9
23.	2	2	.	3	6	5
24.	1	1	.	3	7	2
25.	2	3	.	0	2	1
26.	-	7	2	.	3	
27.	-	3	2	.	7	6
28.	-	3	7	.	9	0
29.	-	2	3	.	1	6
30.	-	0	.	7	9	2

Figure 1: A sample completed answer sheet.

2.2 Detecting the Answer Locations

In order to grade an answer sheet, the answers on the page must first be located. The end goal was to provide automatic detection of the answers on the page while remaining tolerant to potential imperfections in the scans. A variety of computer-vision based approaches were explored to dynamically locate the coordinate locations of the corners of each digit box for every answer in the image.

2.2.1 Template Matching

Initially, the boxes were detected by using the `cv2.matchTemplate()` function to find the individual corners of each of the answer boxes. A corner-shaped kernel was used as a template that was scanned across the scanned image to determine which parts of the scan were likely to be a corner.

This did a reasonable job at detecting all of the answer boxes for most scans, but it still resulted in false positives from the text on the page and often times missed certain corners. This was alleviated by attempting to detect and reject outlier corners while also attempting to repair missing corners by extrapolating from neighbors. This worked reasonably well but wasn't reliable enough to be used as a long-term solution.

2.2.2 Contour Detection

The final implementation ultimately used contour detection and filtering to locate the answer boxes on the page. A contour is defined as the curve following a set of continuous points with the same color along a boundary. In the context of this problem, the contours of interest would follow the interior edges of the boxes on the page.

The `findContours()` function in OpenCV was used to obtain a list of all contours present on the scanned answer sheet. This list of contours would contain the desired contours defining the boundaries of the answer boxes as well as additional irrelevant contours. To isolate the contours following the interior edges of each box, all contours were filtered based on basic shape properties. The `cv2.approxPolyDP()` function was used to translate each contour into a polygon approximation rather than a continuous set of points. Each approximation was filtered to ensure that it contained exactly 4 sides of approximately equal length and also had a sufficiently large area. These conditions were sufficient to robustly detect all unobstructed answer boxes on the page without any false positives.

The main issue with this approach is that it completely fails to detect any boxes in the scan that are partially obstructed. Examples of possible obstructions resulting from print errors or student writing are shown in Figure 2. Any of the obstructed cases in Figure 2 would prevent the algorithm from detecting the square contour.

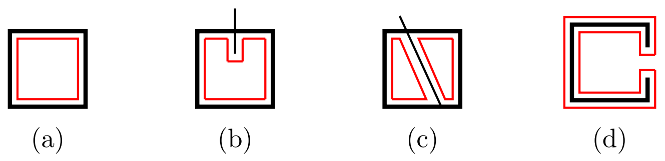


Figure 2: Sample obstructions of answer boxes. The resulting detected contours are also shown in red. The examples include: a) an ideal unobstructed box, b) a partially obstructed box, c) an obstruction connecting two sides, and d) a box with a gap.

To alleviate this, the scans are preprocessed to remove any of these possible obstructions before performing the contour search. The following sequence of morphological operations is performed on each scan:

1. The binary thresholded version of the scan is passed through a morphological dilation operation (`cv2.dilate()`) with a small square kernel to emphasize the lines in the image and fill in any small gaps.
2. A morphological close operation (`cv2.morphologyEx(op=cv2.MORPH_CLOSE)`) is performed with both a horizontal kernel (shape 1 x n) and a vertical kernel (shape n x 1). This fills in any gaps that occur in the edges of the answer boxes such as case (d) in Figure 2.
3. A morphological open operation (`cv2.morphologyEx(op=cv2.MORPH_OPEN)`) is performed with both a horizontal kernel and a vertical kernel. An open operation consists of an erosion followed by a dilation. This removes anything in the image that is not either a vertical or horizontal line. This removes the various characters on the page as well as eliminates any obstructions such as case (c) in Figure 2 that are not perfectly straight in the vertical or horizontal direction.

The exact dimensions of the kernels used in the morphological operations was scaled proportionally to the DPI of the scanned answer sheet. The close and open operations in the horizontal and vertical dimensions were performed independently of each other and later recombined to avoid cross-contamination of artifacts. By performing these preprocessing steps, reliable detection of all test scans was achieved.

2.3 Digit Classification

Once the answer boxes are located, they must be extracted and classified as one of the 12 supported classes: the digits 0-9, a decimal point, or a negative sign. To classify the digits extracted from the answer sheet, the pixel values of the digits were passed through a convolutional neural network built with TensorFlow. The input images were normalized to a size of 28x28 pixels with float32 intensities between 0 and 1.

Additionally, the model accepts a description of the size of the written digit as well as its position within the answer box. This gives the model the chance to more easily differentiate between cases such as a 0 and a decimal place. While a 0 and a decimal point might have similar shapes, a 0 will typically be much larger and a decimal point will typically be written lower in the box. Four additional float32 values between 0 and 1 are appended to the input describing the x and y offset, and the size of the bounding box of the handwritten digit in the x and y directions relative to the answer box.

2.3.1 Model Architecture

The model architecture was heavily based off of the *Deep MNIST for Experts* tutorial on the TensorFlow website [1]. A few modifications were made to better tailor the model to this specific use case. The additional decimal point and negative sign classes were added to support a wider range of values of potential answers. Additionally, the description of the bounding box around the digit was added as additional input data to the model to better differentiate between classes as discussed above.

The input to the model is 788 float32 values which includes the 28x28 pixels of the digit image along with the 4 values describing the size and proposition of the digit. The 784 pixel values of the 28x28 image are passed through an initial 2D convolution layer using a kernel size of 5x5, a stride of 1x1, and 32 output channels. The next layer consists of a second 2D convolution layer with the same kernel and stride parameters but 64 output channels. The next layer acts as a fully connected layer which reduces the results of the convolution layers down to 1024 values. The 4 values describing the size and position of the handwritten digit are directly appended to the 1024 output nodes of the first fully connected layer, bypassing the convolution layers and first fully connected layer. These values are passed through the final fully connected layer which contains 12 output nodes, one for each of the supported classes. The output nodes of the second fully connected layer are run through a softmax operation to determine the respective probabilities of each class.

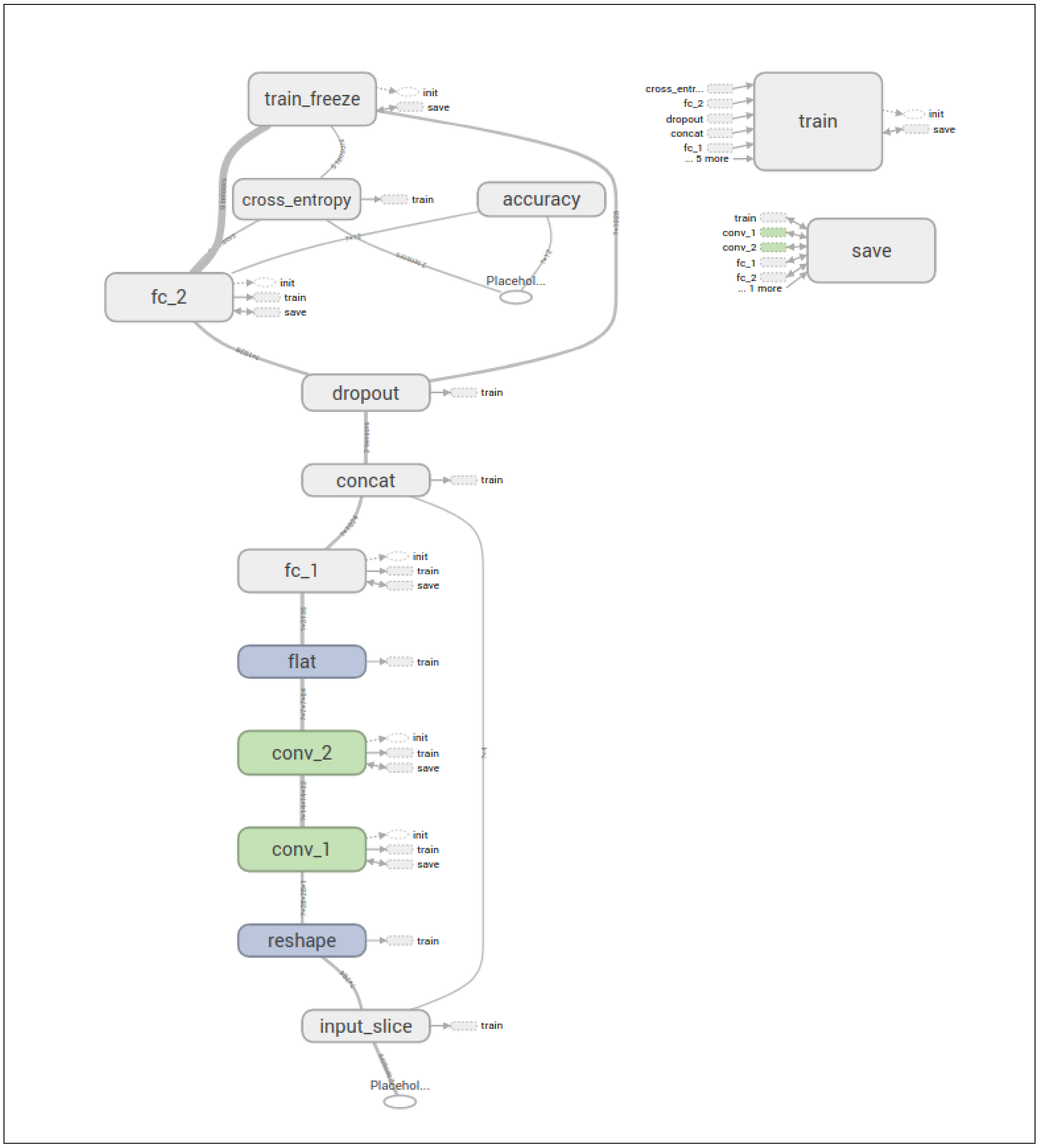


Figure 3: The model architecture visualized in TensorBoard.

2.3.2 Model Training

The majority of the samples used for the training process came from the MNIST handwritten digit dataset. The MNIST dataset consists of approximately 6500 handwritten samples of each digit between 0 and 9. The MNIST dataset lacks samples of decimal points and negative signs so those samples were collected manually. Around 200 samples each of both handwritten decimal points and negative signs were collected from various volunteers. These samples were used to generate a number of samples comparable to the MNIST dataset by applying random image transformations including left-right flips, up-down flips, Gaussian blurs, affine scales, translations, and rotations.

The training process used the `tf.train.AdamOptimizer` optimizer to minimize the softmax cross entropy of the model over 50 epochs of the training data. The training process took 5 minutes on a GTX 980 Ti and resulted in a final accuracy of 0.9901 on the test set. The accuracy of the modified model was slightly lower than the 0.9920 accuracy achieved in the Deep MNIST tutorial [1], but it was still high considering the additional complexity of classifying more classes.

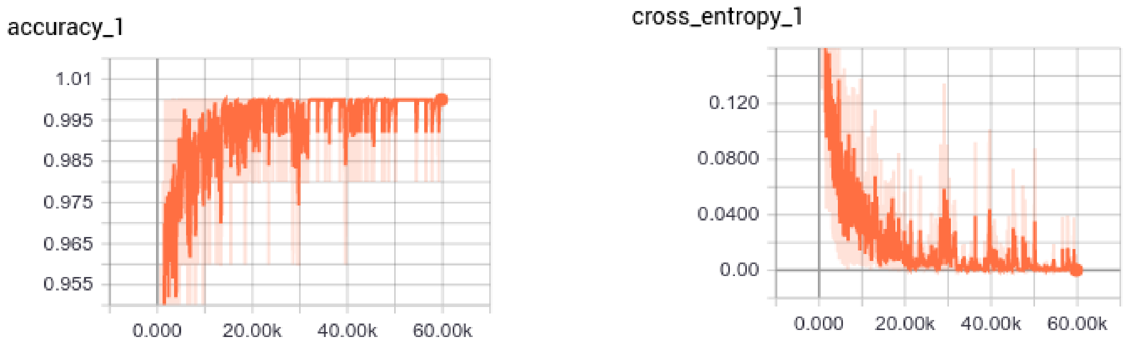


Figure 4: The TensorBoard visualization of the accuracy and cross entropy metrics of the training process.

The model weights calculated during the training are saved as the base model which is restored for future executions of the program.

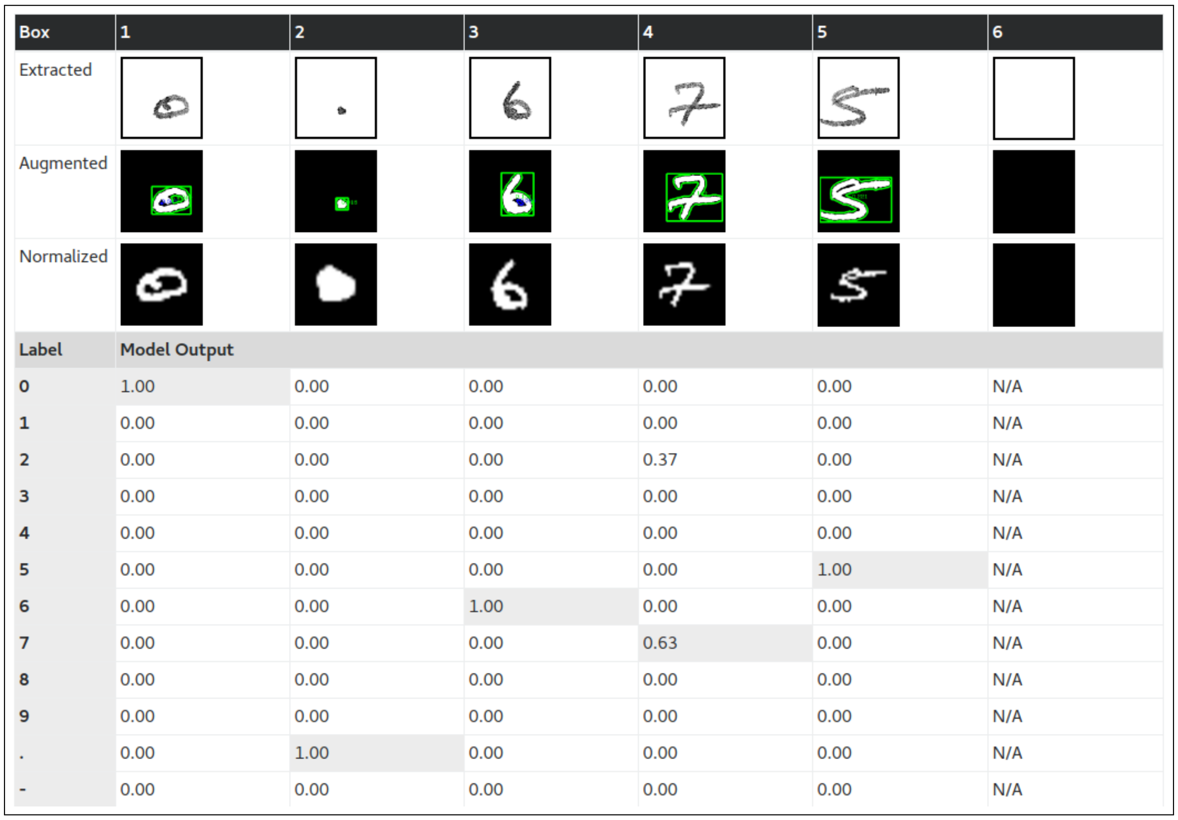
2.3.3 Model Calibration

The base model can be further tailored to specific handwriting by collecting additional handwriting samples from the student. Using as few as 10 samples of each of the digit classes for additional training can significantly improve the prediction accuracy of the model for people with unique handwriting. The intended use case is to collect handwriting samples of known digits from each student at the beginning of the term to use for calibrating unique models for each student that can be used for future grading.

The process restores the original base weights learned during the original training as the starting point. When calibrating the model, all weights are frozen except for the weights of the final fully connected layer. This strategy of transfer learning preserves the important features learned during the original training, but learns to weigh each feature differently between the possible output classes. The final layer's weights are retrained using the collected handwriting samples for 200 epochs due to the low number of samples. The newly learned weights are then saved to disk for future grading.

2.4 Grading

The model output consists of a probability measure between 0 and 1 for all of the digit classes for each of the boxes in the answer. This value indicates how likely it is that the student wrote the respective digit in that box. A sample of the model output can be seen in Figure 5. The grading process uses this output information to determine a score indicating the confidence that the answer is correct given a range of accepted answer values.



Box	1	2	3	4	5	6
Extracted						
Augmented						
Normalized						
Label	**Model Output**					
0	1.00	0.00	0.00	0.00	0.00	N/A
1	0.00	0.00	0.00	0.00	0.00	N/A
2	0.00	0.00	0.00	0.37	0.00	N/A
3	0.00	0.00	0.00	0.00	0.00	N/A
4	0.00	0.00	0.00	0.00	0.00	N/A
5	0.00	0.00	0.00	0.00	1.00	N/A
6	0.00	0.00	1.00	0.00	0.00	N/A
7	0.00	0.00	0.00	0.63	0.00	N/A
8	0.00	0.00	0.00	0.00	0.00	N/A
9	0.00	0.00	0.00	0.00	0.00	N/A
.	0.00	1.00	0.00	0.00	0.00	N/A
-	0.00	0.00	0.00	0.00	0.00	N/A

Figure 5: The model output for a sample answer.

The grading process is complicated because there typically isn't a single allowed answer for numerical questions. Most manual graders will allow for a certain amount of tolerance in the answer or expect a specific number of significant digits. Additionally, there are multiple ways of writing the same numerical answer. For example, a student could write 0.675, .675, or 0.6750, all of which could be considered correct. As a result, the algorithm accepts a range of values that should be considered correct for each answer that is graded.

2.4.1 Direct Grading

The first approach suggested for grading involved directly observing the individual probabilities to determine the most likely combinations of digits that the student wrote. Each of these candidate combinations could be parsed and checked if they are within the range of allowed answers. If one of the candidate combination was within the range of allowed answers, the answer would be considered correct. The main issue with this approach is that the digits that the student wrote are not guaranteed to be one of the highest candidates from the model output. It is entirely possible for the student to write a 6 and have the model predict 6 as the least likely label. The output of the model is only a probabilistic estimate and is prone to error.

In order to guarantee that any potential combination of digits that the student wrote is considered, the algorithm would have to enumerate all d^n combinations where d is the number digit classes (12) and n is the number of boxes in each answer. As a result, the time complexity of this algorithm would grow exponentially with respect to the number of boxes in each answer.

2.4.2 Indirect Grading

The alternative approach to grading involves analyzing the range of accepted values and determining what possible combinations the student could have written in order for the answer to be considered correct. For example, if the allowed range of potential answers is 22.65–22.85, the following patterns of digits would be considered correct:

- 22.6{5-9}*
- 22.7*
- 22.8{0-4}*
- 22.85

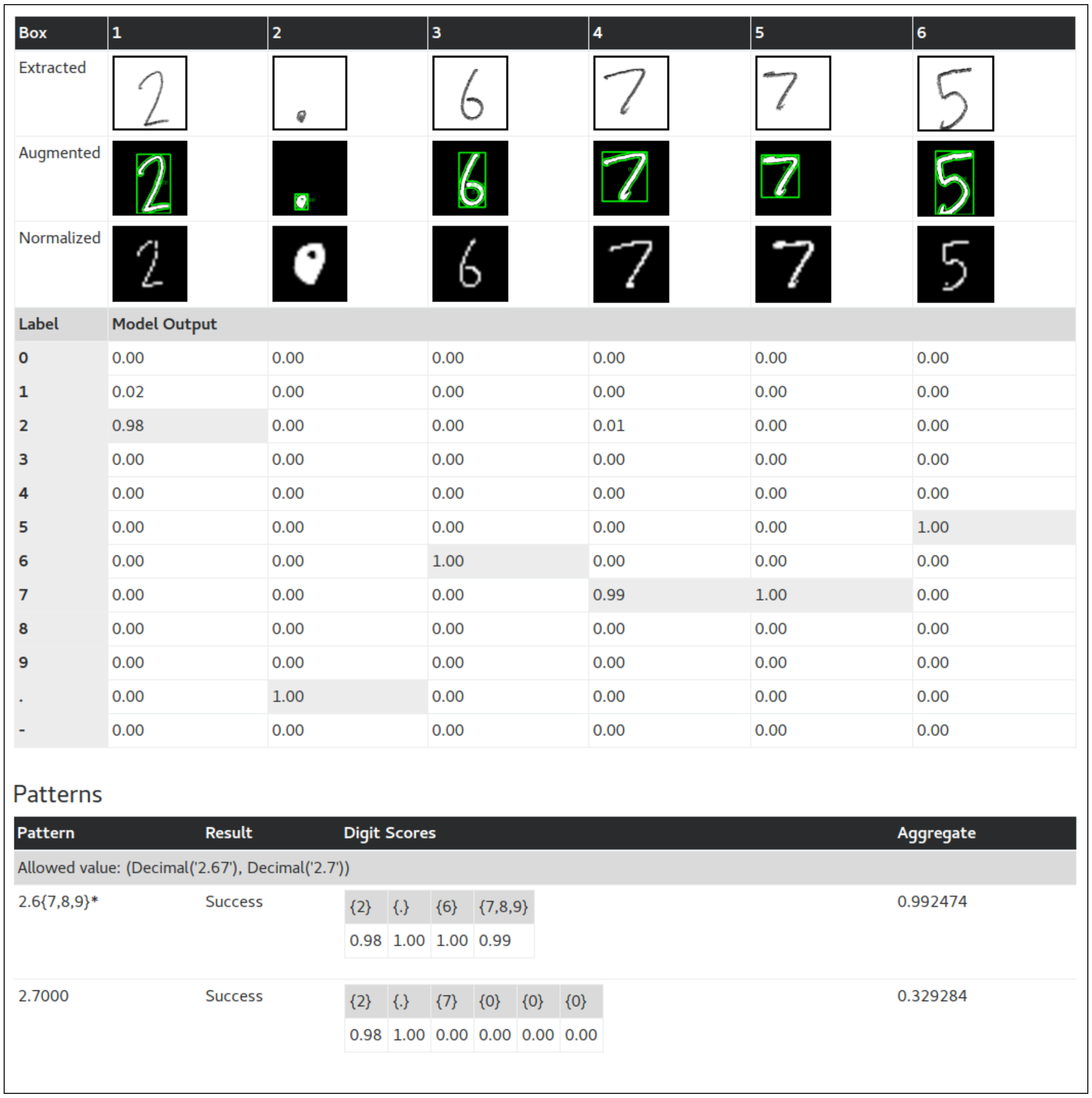
The $\{d_1\text{-}d_2\}$ notation represents a range of interchangeable digits, any one of which would be considered correct, and the * wildcard indicates that any sequence of trailing digits would be considered correct.

The benefit of this approach is that it allows the algorithm to consider multiple possible answers simultaneously while grading a single pattern by considering entire ranges of allowed digits. The algorithm analyzes each digit, range of digits, or wildcard one by one and generates individual digit scores indicating the correctness of a single digit written by a student. When grading, the algorithm handles these cases as follows:

- Single digit: The model output for the specified digit in the pattern is used as the digit score.
- Digit range: The algorithm gives the student the benefit of the doubt and uses the highest model output for any of the digits within the range as the digit score.
- Wildcard: Grading terminates at the wildcard because no future digits are relevant.

Once the individual digit scores are determined, the digit scores are aggregated into a single score. The multiple possible patterns are considered by taking the maximum all of the aggregate outputs for each of the patterns. If this score exceeds a set threshold, the answer is considered correct. The selection of the aggregation function and the threshold is discussed in section 2.5.

This approach is also guaranteed to consider any potential correct answer the student could have written because all correct answers are considered. The ability to consider multiple answers at once using digit ranges allows the algorithm's complexity to grow linearly with respect to the number of boxes in the answer. This approach was used in the final implementation of the project.



Box	1	2	3	4	5	6
Extracted						
Augmented						
Normalized						
Label	**Model Output**					
0	0.00	0.00	0.00	0.00	0.00	0.00
1	0.02	0.00	0.00	0.00	0.00	0.00
2	0.98	0.00	0.00	0.01	0.00	0.00
3	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.00	0.00	1.00
6	0.00	0.00	1.00	0.00	0.00	0.00
7	0.00	0.00	0.00	0.99	1.00	0.00
8	0.00	0.00	0.00	0.00	0.00	0.00
9	0.00	0.00	0.00	0.00	0.00	0.00
.	0.00	1.00	0.00	0.00	0.00	0.00
-	0.00	0.00	0.00	0.00	0.00	0.00

Patterns

Pattern	Result	Digit Scores	Aggregate
Allowed value: (Decimal('2.67'), Decimal('2.7'))			
2.6{7,8,9}*	Success	{2} {.} {6} {7,8,9} / 0.98 1.00 1.00 0.99	0.992474
2.7000	Success	{2} {.} {7} {0} {0} {0} / 0.98 1.00 0.00 0.00 0.00 0.00	0.329284

Figure 6: The detailed output of the indirect grading process. Answers between 2.67 and 2.7 were considered correct. The algorithm assigned a high digit score to the {7, 8, 9} digit range because 7 was scored highly by the model at that position.

2.5 Aggregate and Threshold Selection

The selection of the aggregate function and threshold value is important because it ultimately decides the accuracy of the algorithm's grading results. The final program was used to grade an exam in Professor Sun's DATA 301 class to test the algorithm. The automated grading results were compared to the baseline results of grading the exam manually to analyze the accuracy metrics while selecting the aggregate function and threshold value.

2.5.1 Aggregate Selection

The aggregate function is responsible for taking the individual digit scores from the grading process (section 2.4.2) and producing a single measure of correctness for the answer. A typical aggregate function should produce higher output values when many of the digits are scored highly while penalizing answers that have digits with low scores. Three functions were considered as potential aggregate functions:

- Mean
- Harmonic mean
- Minimum

The precision-recall curves of the model were generated for each aggregate function. In this context, the precision indicates what percentage of answers classified as correct by the algorithm were actually correct, and the recall indicates what percentage of correct answers were classified as correct by the algorithm. A larger area under the precision-recall curve indicates a better classifier.

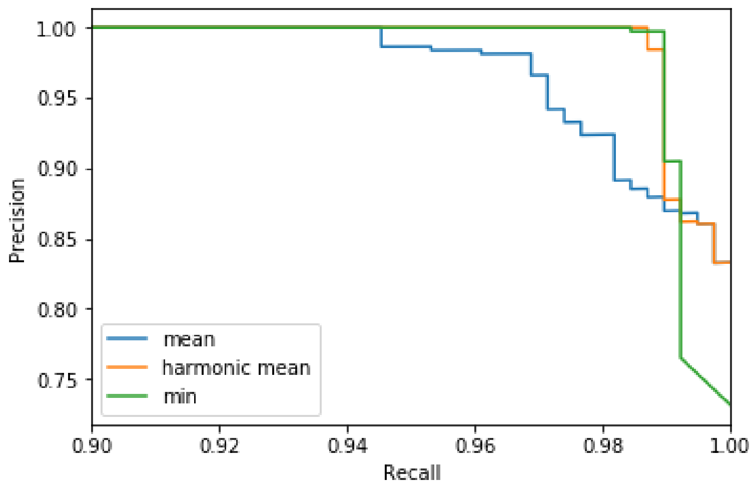


Figure 7: The precision-recall curve of each suggested aggregate function.

Based on the precision-recall curves, the harmonic mean function seems to achieve the same precision as the mean function while also maintaining higher recall. As a result, the harmonic mean function was selected as the default aggregate function.

2.5.2 Threshold Selection

The threshold value is the minimum aggregate output required for an answer to be considered correct. The optimal threshold value was determined by plotting the recall and precision curves as a function of the threshold value using the harmonic mean aggregate.

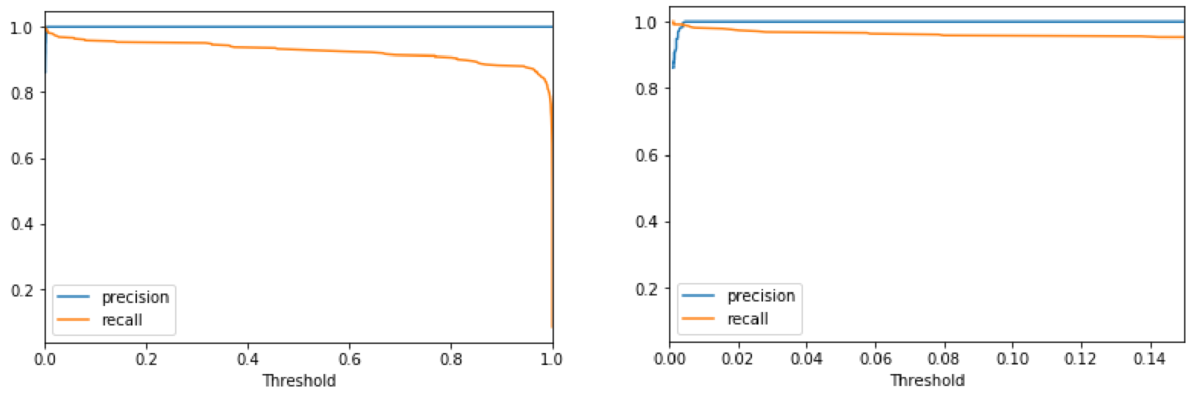


Figure 8: The precision and recall values as a function of the threshold value using the harmonic mean aggregate function.

The precision values appear to be constant at 1.0 for any threshold value greater than 0.01. However, as the threshold value increases, the recall slowly decreases. A threshold value of 0.1 was selected as the default threshold because it comfortably resulted in a precision of 1.0 while also maintaining relatively high recall.

2.6 Interface

The final implementation is used through a command-line interface that allows the professor to grade scanned answer sheets or calibrate new models. These two functionalities are accessed through the subcommands `grade` and `calibrate`, respectively.

2.6.1 Grading Scans

The `grade` subcommand is used to grade scanned answer sheets. A single answer sheet or a directory containing multiple answer sheets can be specified to grade. The detailed usage is shown below.

```
[mark@mark-pc grader]$ python grader.py grade --help
usage: grader.py grade [-h] [--debug] [--report-dir REPORT_DIR]
                       [--model MODEL | --model-prefix]
                       scan info

positional arguments:
  scan                  path to a scan file or a directory containing scans
  info                  path to an info file containing scan properties and
                        the reference key for grading or expected digits for
                        calibration.

optional arguments:
  -h, --help            show this help message and exit
  --debug
  --report-dir REPORT_DIR
                        generate and output reports to this directory
  --model MODEL         specify a specific model to use for grading or as the
```

```
                        target for calibration
  --model-prefix        use the prefix of the scan filenames to determine
                        which model to use
```

A separate info file must also be specified along with the scans. This must be a JSON file that specifies the DPI of the scans, the number of answers on the answer sheets, the number of answers per column, and the number of boxes in each answer.

Most importantly, the correct answers must also be specified in a list associated with the `answers` name. This list contains a series of lists of accepted answers for each answer to be graded. Each accepted answer is either a single scalar value or a 2-element list containing the start and stop values of a range of accepted answers. A sample info file for the `grade` subcommand is shown below.

key.json

```
{
  "name": "sample_quiz",
  "dpi": 600,
  "num_answers": 4,
  "answers_per_col": 2,
  "boxes_per_answer": 6,

  "answers": [
    [[33.32, 33.55]],
    [-1.01],
    [44.6],
    [12, -12]
  ]
}
```

2.6.2 Calibrating Models

The `calibrate` subcommand is used to calibrate custom models for future usage with the `grade` subcommand. A single answer sheet or a directory containing multiple answer sheets can be specified to be used as sources for calibration. The detailed usage is shown below.

```
[mark@mark-pc grader]$ python grader.py calibrate --help
usage: grader.py calibrate [-h] [--debug] [--report-dir REPORT_DIR]
                           [--model MODEL | --model-prefix]
                           scan info

positional arguments:
  scan                  path to a scan file or a directory containing scans
  info                  path to an info file containing scan properties and
                        the reference key for grading or expected digits for
                        calibration.

optional arguments:
  -h, --help            show this help message and exit
  --debug
  --report-dir REPORT_DIR
                        generate and output reports to this directory
  --model MODEL         specify a specific model to use for grading or as the
                        target for calibration
  --model-prefix        use the prefix of the scan filenames to determine
                        which model to use
```

A separate info file must also be specified along with the scans. This must be a JSON file that specifies the DPI of the scans, the number of answers on the answer sheets, the number of answers per column, and the number of boxes in each answer.

The expected digits to be written in each box must be specified in a list associated with the `expected` name. This list contains a series of lists of expected digits for each set of boxes. These values are used as the labels for the training process so it's important that the values exactly match what was written by the students. A sample info file for the `calibrate` subcommand is shown below.

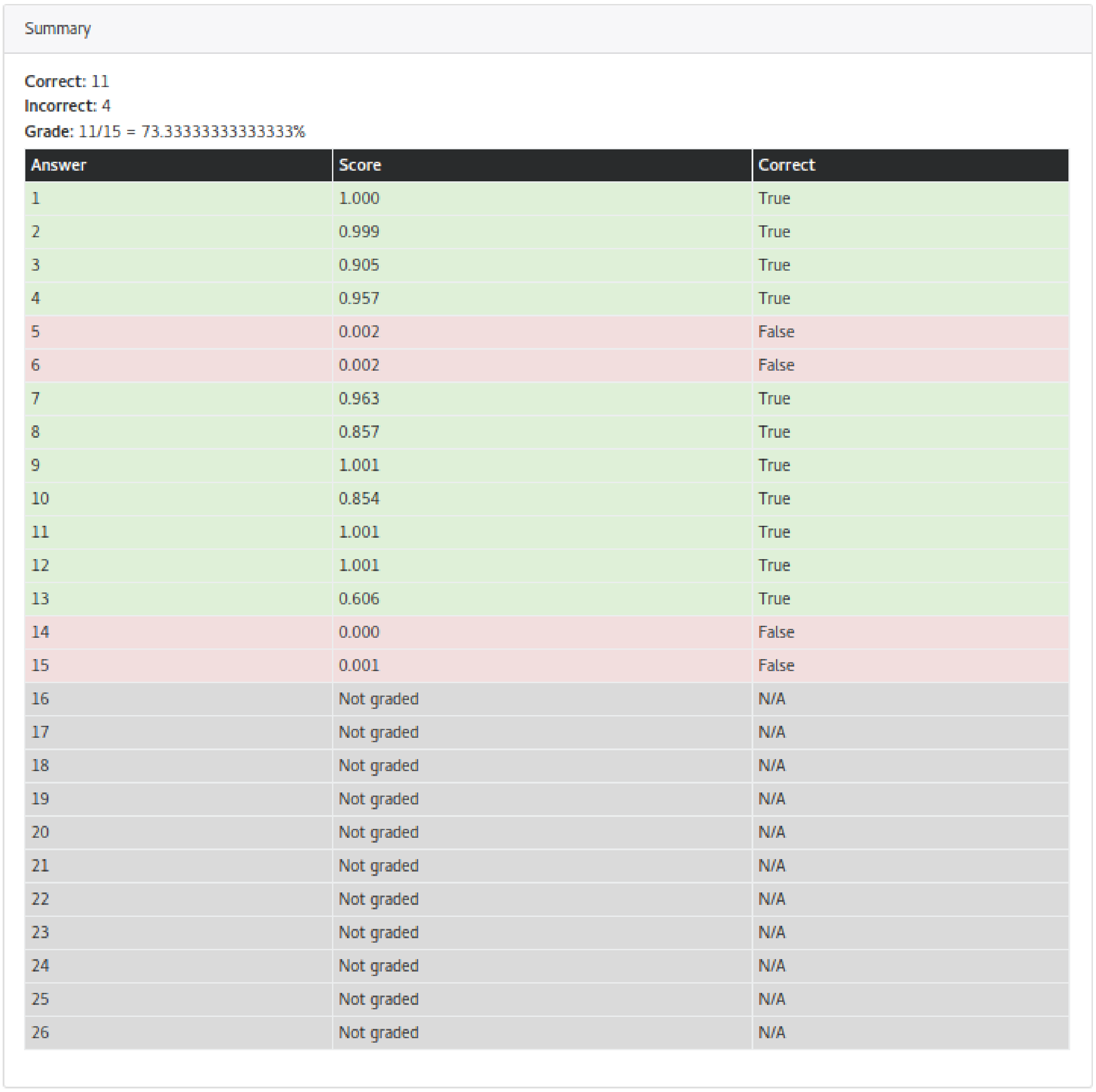
expected.json

```
{
  "name": "sample_expected",
  "dpi": 600,
  "num_answers": 4,
  "answers_per_col": 2,
  "boxes_per_answer": 6,

  "expected": [
    ["-", "7", "3", "3", ".", "5"],
    ["1", "7", "3", "1", "2", "9"],
    ["-", "1", ".", "2", "7", "0"],
    ["5", "0", "8", "8", ".", "1"]
  ]
}
```

2.6.3 Report

An optional report can be generated by the program using the `--report-dir` command line argument to specify the directory to save reports in. The report includes a summary section along with a detailed view of every answer that was processed. The detailed view includes an image of every digit that was extracted along with the model's predicted probabilities of each label. Additionally, the accepted answers are listed along with the possible digit patterns that would be considered correct. The aggregate scores of each digit patterns are shown as well as the final score of the answer. This allows the user to understand the reasoning behind the score given for the answer.

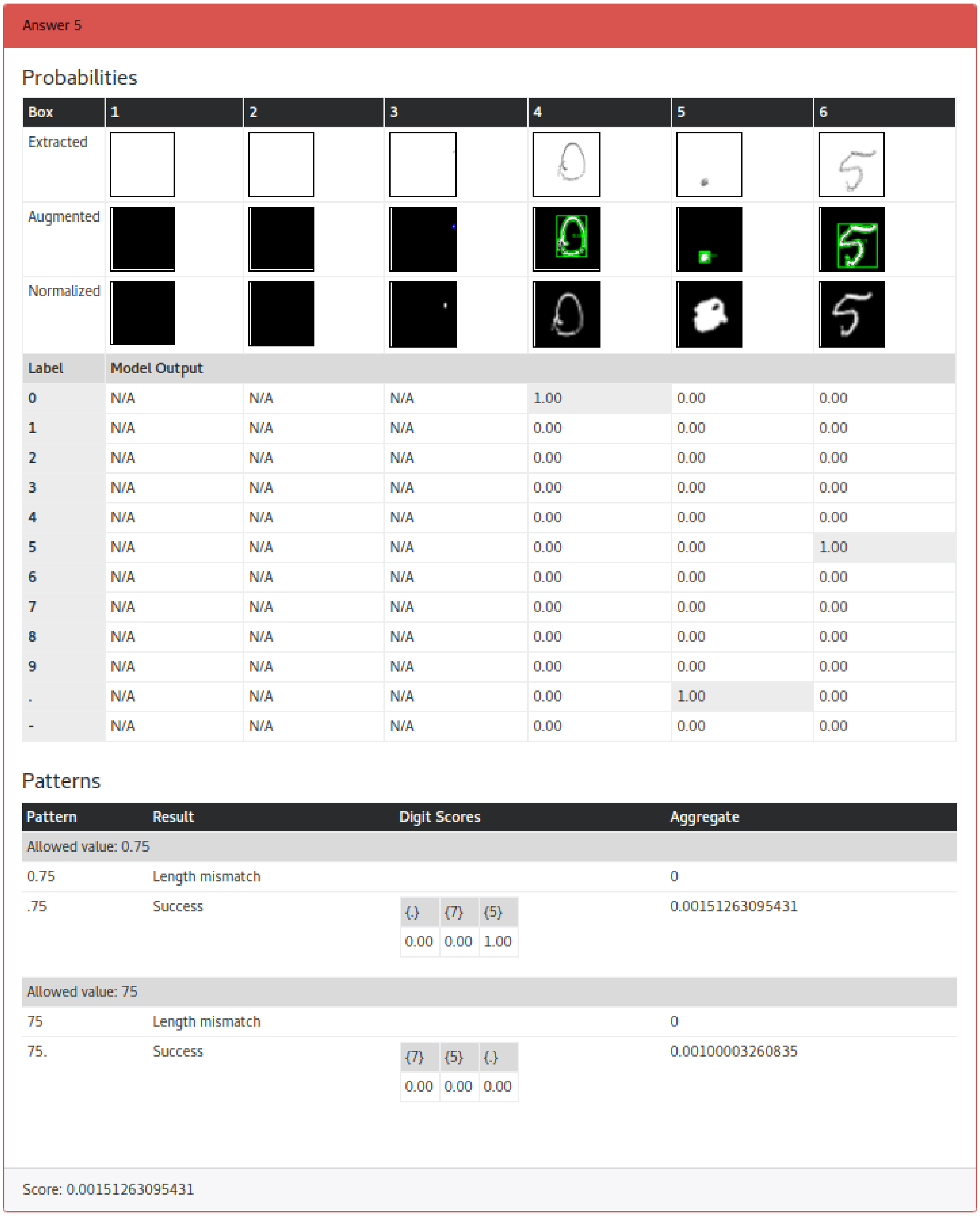


Summary

Correct: 11
Incorrect: 4
Grade: 11/15 = 73.33333333333333%

Answer	Score	Correct
1	1.000	True
2	0.999	True
3	0.905	True
4	0.957	True
5	0.002	False
6	0.002	False
7	0.963	True
8	0.857	True
9	1.001	True
10	0.854	True
11	1.001	True
12	1.001	True
13	0.606	True
14	0.000	False
15	0.001	False
16	Not graded	N/A
17	Not graded	N/A
18	Not graded	N/A
19	Not graded	N/A
20	Not graded	N/A
21	Not graded	N/A
22	Not graded	N/A
23	Not graded	N/A
24	Not graded	N/A
25	Not graded	N/A
26	Not graded	N/A

Figure 9: A sample summary section of a generated report.



Answer 5

Probabilities

Box	1	2	3	4	5	6
Extracted						
Augmented						
Normalized						
Label	**Model Output**					
0	N/A	N/A	N/A	1.00	0.00	0.00
1	N/A	N/A	N/A	0.00	0.00	0.00
2	N/A	N/A	N/A	0.00	0.00	0.00
3	N/A	N/A	N/A	0.00	0.00	0.00
4	N/A	N/A	N/A	0.00	0.00	0.00
5	N/A	N/A	N/A	0.00	0.00	1.00
6	N/A	N/A	N/A	0.00	0.00	0.00
7	N/A	N/A	N/A	0.00	0.00	0.00
8	N/A	N/A	N/A	0.00	0.00	0.00
9	N/A	N/A	N/A	0.00	0.00	0.00
.	N/A	N/A	N/A	0.00	1.00	0.00
-	N/A	N/A	N/A	0.00	0.00	0.00

Patterns

Pattern	Result	Digit Scores	Aggregate
Allowed value: 0.75			
0.75	Length mismatch		0
.75	Success	{.} {7} {5} / 0.00 0.00 1.00	0.00151263095431
Allowed value: 75			
75	Length mismatch		0
75.	Success	{7} {5} {.} / 0.00 0.00 0.00	0.00100003260835

Score: 0.00151263095431

Figure 10: A sample detailed view of an incorrect answer in a report.

3 Results

The final program was tested by using it to grade a 15-question exam taken by 35 students. The exam consisted of a mixture of multiple choice questions and numerical response questions. Sample handwriting data was collected from each student prior to the exam to calibrate custom models for each student. The automated grader's classification (correct/incorrect) was compared to the manual grading results to determine the accuracy metrics. The program was configured to use the default harmonic mean aggregate function with a threshold of 0.1.

Accuracy	0.9676
Precision	1.0000
Recall	0.9557
F1 Score	0.9774

(a) Calibrated models

Accuracy	0.9524
Precision	1.0000
Recall	0.9349
F1 Score	0.9664

(b) Uncalibrated base model

Figure 11: Accuracy metrics of the final program using (a) calibrated models, or (b) using the uncalibrated base model.

These results show that every answer marked as correct by the algorithm was actually correct. However, only 95.6% of correct answers given by students were successfully automatically graded as correct. This suggests that the algorithm is occasionally marking correct student answers as incorrect. The following factors were identified as the most common causes of these false negatives:

- Stray markings in the answer boxes not being rejected and throwing off the digit comparisons in the pattern grading process.
- The model confusing decimal points with 1s when written as commas rather than circular points.
- Light writing not being segmented correctly when the handwritten digit is extracted.

The 0.9557 recall metric is promising, but shows that the program in its current state needs to be verified by a human to achieve completely accurate grading. If nothing else, the program is able to significantly reduce the workload of the grader by only requiring the grader to double check answers classified as incorrect by the algorithm rather than every answer. However, the main causes of the false negative rate don't seem impossible to overcome and future improvements will hopefully bring the recall rate up to an acceptable level.

3.1 Future Improvements

A few potential areas of improvement have been identified for future work on the project.

Model Architecture

The current model struggles with distinguishing between certain pairs of classes such as the decimal points and 0, and negative signs and 7. These should be easily distinguishable by the size and position of the handwritten digit in the answer box. This might improve if the model is modified to add an additional fully connected layer which attempts to emphasize the importance of the size and position features relative to the actual digit image.

Grading

Currently, the recall rate can be improved by lowering the threshold value. However, lowering the recall rate further leaves very little room for model classification error for single digit answers. If the grading algorithm was modified to support a dynamic threshold based on the number of digits in the answer, it might be possible to achieve the best of both worlds.

4 Appendix

4.1 Source Code

The source code for the project can be found at `https://github.com/mtbrown/grader`.

4.2 Senior Project Questionnaire

Summary of Functional Requirements

The final implementation of the project allows a professor to automate the grading process for numerical answers on tests. The program automatically detects the answers on a scanned answer sheet and compares them to the correct answers specified by the professor. An algorithm generates a score indicating the confidence in the correctness of each answer. This information is used to classify each answer as correct or incorrect and provide an overall grade.

Primary Constraints

This project was made difficult by the inconsistency of inputs into the system. When detecting the locations of answers on the scanned answer sheet, the algorithm must account for various errors such as misuse by students, errors during printing, or variations between different scanners. Additionally, the variation in student handwriting makes it difficult to reliably classify all digits with high accuracy. The grading algorithm must allow for tolerance in its inputs to account for potential misclassifications by the digit prediction model.

Economic

This project was software-based so there were no costs associated with development. A scanner is required to make full use of the project and the associated cost will vary.

The estimated development time was 6 months and the project was successfully completed within 6 months.

This will not be manufactured on a commercial basis. The software will be open-sourced and available for anyone to use.

Environmental

The use of this project will not have any significant environmental impacts.

Manufacturability

This project will not be manufactured.

Sustainability

The project will be open-sourced so releasing updates should be a straightforward process. Possible improvements that have been suggested include modifying the digit classification model to further emphasize the size and position features to improve the classification of decimal places, and improving the grading algorithm to be more strict while grading single digit answers.

Ethical

Overuse of this project could result in too much emphasis on the correctness of a student's final answer rather than the process taken to get there. Automating the grading process also prevents professors from providing helpful feedback as they are grading.

The open nature of the project could also lead to students attempting to find exploits in the grading algorithm to gain an unfair advantage.

Health and Safety

There aren't any health or safety concerns associated with the project.

Social and Political

There aren't any social or political concerns associated with the project.

Development

The development of this project required many new skills that were learned during the course of the project. Newly learned computer vision techniques were used to parse and automatically locate the answers on the answer sheets. Also, data science and machine learning skills were required to design the digit prediction model.

References

[1] Deep MNIST for Experts. (n.d.). Retrieved June 13, 2017, from `https://www.tensorflow.org/get_started/mnist/pros`

[2] L. D. Jackel, D. Sharman, C. E. Stenard, B. I. Strom and D. Zuckert, "Optical character recognition for self-service banking," in AT&T Technical Journal, vol. 74, no. 4, pp. 16-24, July-Aug. 1995. doi: 10.1002/j.1538-7305.1995.tb00189.x, from `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6770173&isnumber=6770170`