Classifying Pen-Based Handwritten Characters Using Neural Networks

Pramoda Karnati
Massachusetts Institute of Technology
pkarnati@mit.edu

Gabe Madonna
Massachusetts Institute of Technology
gmadonna@mit.edu

Neha Prasad
Massachusetts Institute of Technology
nehap@mit.edu

***Abstract*— Online handwriting classification is a challenge with enormous potential but little study compared to its offline counterpart. Existing literature focuses on nearest neighbor approaches, but in this work we propose a Neural Network model (specifically, a Recurrent Neural Network as well as a Convolutional Neural Network) in order to classify two datasets of temporal character coordinate sequences. Our approaches were able to obtain an accuracy of 99.55% on our first dataset and 79.29% on the second, more complex dataset. What we found was that the accuracies of our models were comparable to that achieved by the more standard nearest neighbor approach.**

I. INTRODUCTION

Handwriting recognition is an important challenge in machine learning for both its applicability and difficulty. It is far from solved enough to be deployed for robust general application, but every improvement in this field brings us closer to putting handwriting on equal footing with typing in the world of digital information recording.

Text classification can be approached as either on-line or off-line character recognition. The off-line variant involves capturing pre-written characters through an optical scanner and classifying the characters from the matrix of the resulting image. This means off-line character recognition applies to any scenario in which we must process external text, be it automated scanned document processing or signboard translation. Neural networks have achieved high accuracy and robustness in the offline classification of typed text.

On the other hand, on-line hand-written character recognition is a more difficult problem but is becoming increasingly tractable and applicable. Pen-based input is fast and flexible but increasingly archaic in the digital era. Technology which can robustly bring it into the modern age is in high demand as tools such as laptop-computers and tablets continue to dominate the world of productivity and information recording. As a result, character classification based on digital hand-written characters has become an important research topic. Integration of robust written text recognition into these familiar products will allow for the speed and universality of handwritten text to finally enjoy the robustness and versatility of digitally recorded information.

For our project, we were interested in exploring on-line character recognition on a database of handwritten characters. On-line systems take two-dimensional (x, y) coordinates of successive points as a function of time and classify characters based on this time-series data. On-line methods are generally superior in terms of accuracy due to the availability of this temporal information of the data.

In this work, we propose various methods of classifying pen-based handwritten characters. Previous works have generally focused on using k-nearest neighbors classifiers, but in our work we propose taking advantage of temporal information of the coordinate data by applying a Recurrent Neural Network (RNN) for character classification. As discussed further in the paper, this method generally achieved high accuracy on the dataset. We also propose the usage of a Convolutional Neural Network (CNN), which still outperforms previous works but not by as much as the RNN.

We used template matching as a classical, intuitive baseline and a k-nearest neighbors classifier as a literature standard baseline to gauge the effectiveness of our models.

The rest of the paper is organized as follows: In Section II, we describe related works in literature and the algorithms applied. Section III contains a description of the dataset as well as pre-processing methods. In Section IV, we describe our two proposed models for pen-based character recognition, with results from Section V used as baseline metrics. We include our results in Section VI and conclusions in Section VII. We also include our interactive application to predict and visualize pen-based characters using our two models in Section VIII. The final section presents future work and contribution.

II. RELATED WORKS

Previous works in recent literature used nearest neighbors classifiers for handwritten character recognition. Connell and Jain (1999)[1] propose two classification methods in their work: nearest neighbor and decision trees; with this approach, they were able to achieve an error rate of 13.1%.

Llorens et al. (2008)[2] described another such approach in their work. Using a k-NN classification scheme based on approximate Dynamic Time Warping (DTW), they are able to achieve an error rate of 10.9% on the UJI Pen Characters dataset.

Prat et al. (2009)[3] designed and built an open-source recognition engine for isolated on-line character recognition. The work proposes a similar approach: k-NN classification using DTW-based comparisons and achieved a classification error of 10.85%.

In addition to such implementations, some literature also suggests usage of Hidden Markov Models to represent the character data for classification. Alenius et al. (2017)[4] describe one such approach, which uses a two-stage HMM classification scheme: for first a character classifier and then a word classifier. This approach worked fairly well and has

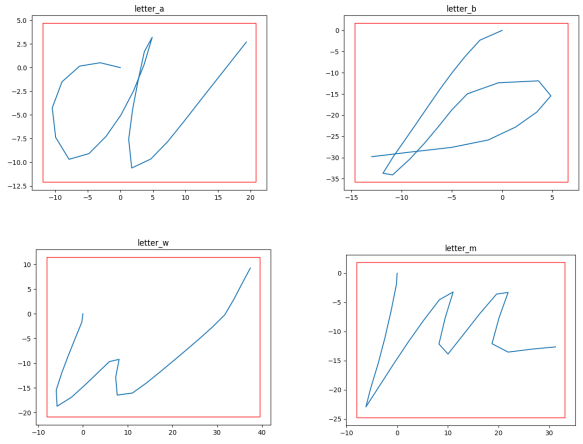

Fig. 1: Dataset 1 Samples

achieved significant results. Various other works propose the use of HMM models and have achieved up to 94% accuracy (6% error) on their specified dataset.

III. DATASET

We used data from the UCI Machine Learning Repository[5]. The first dataset we used, called Dataset 1 for this paper, was the Character Trajectories Data Set [7]. Due to the small number of classes in this dataset, it became clear that testing our hypotheses about good classification techniques would require a dataset with more classes (the idea being that a more challenging dataset would mean a starker contrast between the results of our methods and conventional methods, allowing us to make more definitive conclusions). This led us to Dataset 2, another dataset in the UCI Machine Learning Repository, the UJI Pen Characters Dataset[6].

A. Dataset 1

Dataset 1 was smaller, comprising 2858 total letter samples. Each sample consisted of time series data describing pen trajectory in character writing along three dimensions: x position, y position, and force magnitude. These coordinates (and force values) were sampled on a WACOM tablet at 200 Hz. The collectors of the dataset opted to only collect the twenty lowercase English alphabet letters which comprised exactly one stroke, meaning no letters where a pen would conventionally be lifted in order to draw the full letter (for example, the letter t was omitted because it is comprised of both a vertical stroke and a horizontal stroke, whereas the letter a is included because it is written in a single motion). In addition, the collectors of the dataset normalized the data, centering each letter at the origin and scaling the letters to have unit standard deviation of coordinate components. Refer to Figure 1 for samples from this dataset with bounding boxes.

B. Dataset 2

Dataset 2 was larger, comprising 11640 total letter samples. Each sample also consisted of time series data describing pen trajectory, but this time along two dimensions: x position and y position. These coordinates were collected at a much lower (unspecified) frequency which resulted in there being around twenty coordinates sampled per letter (as opposed to over one hundred per letter for Dataset 1). For this dataset, the collectors of the dataset collected all twenty-six English alphabet letters, and in both upper- and lowercase. Additionally, this dataset was not normalized in space.

C. Pre-Processing

For dataset 1, pre-processing meant first spline-interpolating the data and resampling in time to yield a letter with n datapoints, where n was held constant for the resampling of all letters. This meant we could represent each letter as an n matrix (for three columns (x, y, force) of n datapoints), which allowed us to package the data into a single tensor for training an RNN and CNN. The number of samples n was modulated from ten to thirty to test how the performance (and training time) of the models responded to the granularity of the data. With too few points (too small n) we knew that models would train quickly but would be inaccurate due to insufficient data for differentiating between samples. With too many points we knew that models would be slow to train and have more data than necessary to look at, reducing the robustness of the models against over-fitting. The happy medium turned out to be around fifteen datapoints. In addition, we experimented with dropping columns and classifying based on only one or two dimensions of data. What we found was that dropping the x or y column drastically reduced the performance of the models, while dropping the force column made no discernible difference. As a result, we dropped the force column in favor of consistency with the dimensions of Dataset 2.

For Dataset 2, pre-processing also meant spline-interpolating the data and resampling in time to yield a letter with n data points. Before we could do this however, we needed to normalize the dataset in space. We decided to standard normalize it, bringing the center of each letter

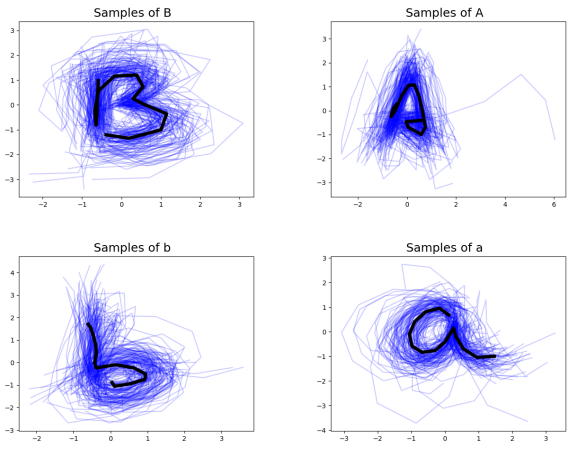

Fig. 2: Dataset 2 Samples with Averages

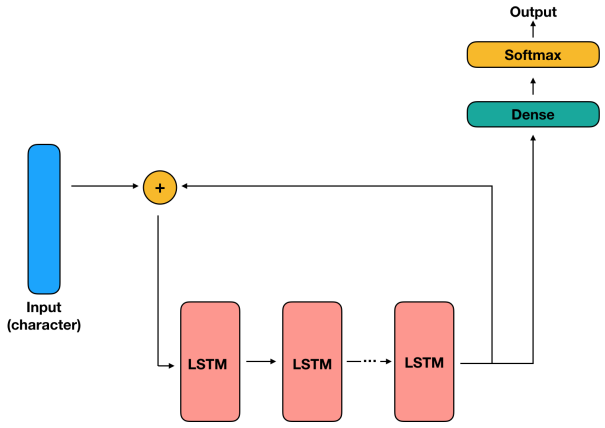

Fig. 3: Proposed RNN architecture

to the origin and re-scaling every letter about the origin to achieve a dataset-wide unit standard deviation in coordinate components (meaning if you made a set of all the x and y coordinate components once the letters were centered at the origin, their standard deviation would be unity).

IV. METHODS

In this work, we propose the use of neural network classification schemes to classify time-series pen character representations. Due to the temporal nature of the sample representations, we hypothesized that a Recurrent Neural Network (RNN) would classify them with high accuracy. As the characters were represented by matrices of coordinates, we also evaluated the performance of a Convolutional Neural Network (CNN), hypothesizing that it might learn spacial patterns as sub-components of letters. Each model is described in detail below.

A. Recurrent Neural Network (RNN)

The overall architecture of an RNN model is described in Figure 3. Each character input is represented as an $n \times 2$ matrix of coordinates where n is the number of coordinates to which each character is resampled. Each layer in the network contains a specified number of LSTM units. The number of layers and the number of LSTM units are hyperparameters which we tuned in testing to see which combination yielded the optimal balance of being lightweight and having adequate complexity. According to RNN convention, for a given letter/sample, we pass in one datapoint (x-y coordinate) per cycle through the LSTM layers, maintaining a latent state vector throughout all cycles which is updated as a linear combination of itself and the output of each cycle at the end of each cycle. After all points are fed in, the resulting state vector goes through a final dense layer which maps the input to a vector with length equal to the number of possible classes. This allows the network to apply Softmax activation, which is a vector of confidences across the different classes (ie if .5 is in element 7 of the output vector, then the model believes there to be a .5 chance that the input belonged to class 7). To determine a predicted class, we simply take the argmax of this vector. We built the model using the Keras[9] library and trained it with categorical crossentropy loss and the adam optimizer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 25, 50)            10800
_________________________________________________________________
flatten_1 (Flatten)          (None, 1250)              0
_________________________________________________________________
dense_1 (Dense)              (None, 20)                25020
_________________________________________________________________
activation_1 (Activation)    (None, 20)                0
=================================================================
Total params: 35,820
Trainable params: 35,820
Non-trainable params: 0
_________________________________________________________________
None
```

Fig. 4: Example RNN Architecture

B. Convolutional Neural Network (CNN)

We also proposed the use of a CNN, an example of which is described in Figure 4. The character data is represented as matrices of coordinates so we treated the matrices as one-channel "images" and classified them with a CNN. According to CNN convention, sample matrices were passed through convolutional layers with $ReLU$ activation. These layers allowed the CNN to detect high level patterns in letter substructure. The resulting pattern representations were then passed through two dense layers so they could interpret those patterns. Finally, Softmax activation was applied to the resulting vector, the argmax of which was taken as the predicted class. Again, all models were built in Keras and trained with the adam optimizer and categorical crossentropy loss.

V. BASELINE

A. Template Matching

To obtain a baseline for our model, we implemented a simple template-matching algorithm on the character dataset. For each character, all of the coordinate matrix samples for that character were averaged to obtain a unique template matrix for that character. Each new sample is then compared

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 13, 2, 32)         320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 13, 2, 64)         18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 2, 128)        73856
_________________________________________________________________
flatten_1 (Flatten)          (None, 3328)              0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               332900
_________________________________________________________________
dense_2 (Dense)              (None, 20)                2020
=================================================================
Total params: 427,592
Trainable params: 427,592
Non-trainable params: 0
_________________________________________________________________
None
```

Fig. 5: Example CNN Architecture

by some distance metric to the templates and classified as the same character with the template to which it was "closest". We explored various metrics to find the one which was most suited to this application. A summary of each distance metric and its corresponding equation can be found in Table I.

TABLE I: Distance Metric Equations

Metric	**Equation**
Eucledian	$\sqrt{\sum(u_i - v_i)^2}$
Standardizd Euclidean	$\sqrt{\frac{\sum(u_i - v_i)^2}{V[x_i]}}$
Squared Euclidean	$\|\|u - v\|\|_2^2$
Cosine	$1 - \frac{u \cdot v}{\|\|u\|\|_2 \|\|v\|\|_2}$
Correlation	$1 - \frac{(u - \overline{u}) \cdot (v - \overline{v})}{\|\|u - \overline{u}\|\|_2 \|\|v - \overline{v}\|\|_2}$
Chebyshev	$\max_i \|u_i - v_i\|$
Canberra	$\sum_i \frac{\|u_i - v_i\|}{\|u_i\| + \|v_i\|}$
Mahalanobis	$\frac{\sum_i \|u_i - v_i\|}{\sum_i \|u_i + v_i\|}$

We tested this template-matching algorithm on the datasets and found that the accuracy varied significantly depending on the distance metric used. We can see the results below in Table II for each of the two datasets. Notice that standardized Euclidean distance was consistently most effective, where $V[x_i]$ is the variance vector over all the i^{th} components of the points, and u_i and v_i correspond to the points of the matrix.

TABLE II: Accuracy by Distance Metric

Metric	**Dataset 1**	**Dataset 2**
Eucledian	0.3433	0.0061
Standardized Euclidean	0.7079	0.2232
Squared Euclidean	0.3185	0.0061
Cosine	0.2318	0.0411
Correlation	0.2672	0.0271
Chebyshev	0.3681	0.0061
Canberra	0.3185	0.1033
Mahalanobis	0.6566	0.1970

For Dataset 1, we can see that the standardized Euclidean distance metric achieved 70.8% accuracy while in the more complex Dataset 2 it could only achieve 22.3% accuracy. Later we will see that the other classification approaches performed far better than these baselines.

B. k-Nearest Neighbors

In order to obtain another baseline for our model, we implemented a k-Nearest Neighbors algorithm. Most of the literature we read employed a k-NN approach, so we implemented one ourselves to compare our neural networks to the standard approach as a baseline. For each sample, we concatenated the columns (already normalized to have n points each) into a single vector of y values followed by x values. This vector representation allowed us to use the Scikit-Learn library[8] to train and test our own k-NN classifier. Refer to Table III to see the performance of the nearest neighbors classifier across different k values. We found that looking at no more than three nearest neighbors yielded good results, with the optimal number of neighbors being one. We see the k-NN accuracy was generally inversely proportionally related to the number of neighbors k, with optimal accuracy achieved for 1 neighbor. We see this yielded 0.9840 accuracy for Dataset 1 and 0.7771 accuracy for Dataset 2.

TABLE III: k-NN Accuracy vs. k

Neighbors	**Dataset 1**	**Dataset 2**
1	0.9840	0.7771
2	0.9540	0.6078
3	0.9805	0.7318
4	0.9699	0.6288
5	0.9752	0.7139
6	0.9681	0.6264
7	0.9717	0.6912
8	0.9628	0.6280

VI. RESULTS

Our RNN model was able to perform comparably well to previous approaches while our CNN was slightly less effective. Using an RNN, we were able to achieve an accuracy of 99.27% on the test set using Dataset 1 and 79.29% on Dataset 2. Both values are slightly higher than those achieved by a k-NN. The optimal RNN for Dataset 1 had one layer of twenty five unit and was trained over fifty epochs. The optimal RNN for Dataset 2 had two layers of twenty five units each and was trained over one hundred twenty-five epochs. The CNN was comparably effective on Dataset 1 but non-negligabley less effective on Dataset 2, achieving an accuracy of 98.58% on Dataset 1 and 71.34% on Dataset 2.

We see in Figure 6 a summary of the results discussed thus far - on Dataset 1, we find that our neural networks perform similarly well to the literature baseline and far better than the classical baseline (74%). We hypothesized that this was because all of the characters are lowercase and composed of a single stroke. In other words, we think the small number of characters and high variance in shape between characters made it easy to differentiate between them. On Dataset 1, we have baseline of 74% accuracy.

After moving to Dataset 2 we see a more interesting comparison between model performances. First, clearly this dataset is not so easy to separate — the baseline accuracy dropped to 23.24% while the highest achieved accuracy dropped to just under (80%). Additionally, notice that the CNN is no longer able to perform as well as the RNN and k-NN models, likely due to the fact that matrix representations of time series data simply lends itself to a temporal model (such as an RNN) much more than to a spatial model (such as a CNN). Meanwhile, the RNN and k-NN models outperform

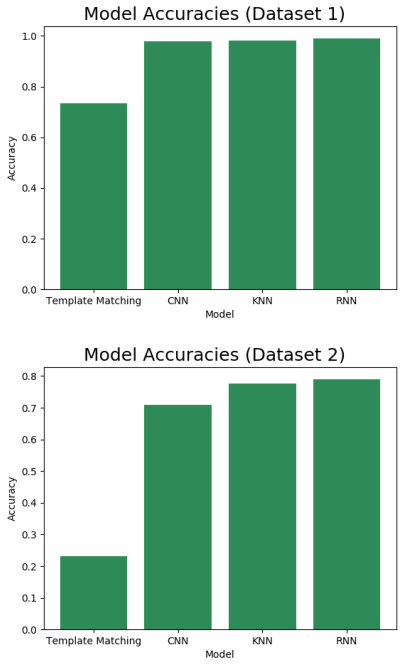

Fig. 6: Accuracy by Model

the other methods and achieve comparable accuracies of 79.29% and 77.71%, respectively. In conclusion, we see that on these two datasets there is no discernible difference between the effectiveness of the RNN and k-NN classifiers.

In Figure 7 we can also analyze the error by letter, looking at what percentage of each letter was incorrectly labeled (we only looked at Dataset 2 here as there was too little classification error in Dataset 1 to get an interesting graph). Here we see the results of what we call the "case invariance" in Dataset 2. Notice that letters such as C, O, K, Z have large error regardless of model while letters such as e, A, f, j have low error, again regardless of model. This uneven distribution of error suggests that there is specific structural similarity in the characters which actually cause certain characters to be harder to classify than others, independent of the model used - we cite case invariance as a particular form of structural similarity which contributed to this phenomenon. We define case invariance as the property whereby characters do not change significantly between their upper- and lower-case representations (as explained earlier, this means letters such as O and C have high case invariance while letters such as A and E have low case invariance). We do not think case invariance alone could be responsible for the low model effectiveness on Dataset 2 however; we think it is this case invariance inherent to the English alphabet combined with unpreserved relative scale in Dataset 2 that caused the classifiers to struggle so much. More precisely, this unpreserved scale refers to how the data was collected in such a way that all letters were about the same size, even before any re-scaling could be performed. This is fine when it means that A is the same size as a as these are not similar to each other (or to any other letter) but it meant that C and

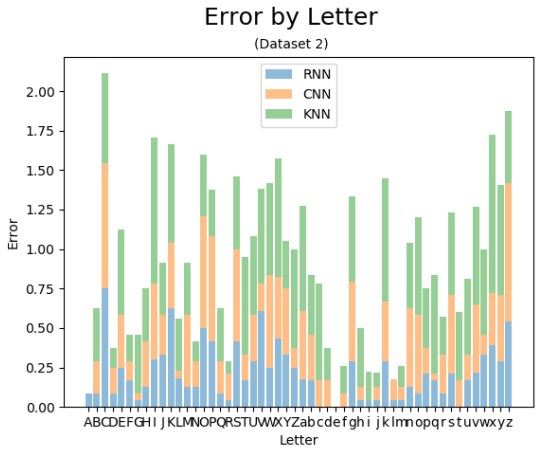

Fig. 7: Error by Character

c, for example, were the same size and thus impossible to differentiate. We see that it is precisely the letters with high case invariance which would be most difficult to classify under this data collection scheme. Their upper- and lower-case representations would be identical - we could not expect much more than 50% accuracy on either representation of these case invariant letters. In conclusion, we attribute most of the error in the models on Dataset 2 to the case invariance inherent to the English alphabet combined with unpreserved relative sizes in the way Dataset 2 was collected.

To assess the performance of both the RNN and the CNN on the two datasets, we tuned various hyperparameters to find the optimal architecture. These experiments are detailed in the following figures. We first varied the number of epochs across the various models. In Figure 8, specifically for Dataset 2, we see that as the number of epochs increased, the accuracy converged to 80%. Beyond 100 epochs, the RNN tended to overfit to the training set, but the test accuracy stayed consistently near 80%. A similar result arose using Dataset 1, but the accuracy converged to a much higher 99.5%.

We now turn to the number of segments n to which each

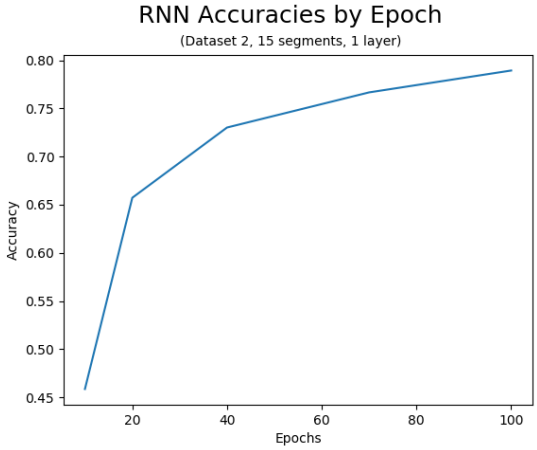

Fig. 8: RNN Accuracy History

character was resampled. In Figure 9, we can see the effect of varying n on the accuracy and runtime. Both the CNN and the k-NN approaches peaked in accuracy for 15 line segments per character, which suggests that the effectiveness of these models actually goes down once we collect too many points in a letter, likely because it makes it harder for the models to work solely on the large scale patterns of each letter rather than less important small scale patterns. An interesting observation is that the RNN and Template Matching accuracies are actually slightly directly correlated with n. A possible explanation is that certain characters originally contain fewer segments than others, but the re-sampling function we used in Scikit-Learn was only able to resample the letters with more than n datapoints already. Therefore, as we increased n we actually began to narrow down the number of characters that were available to the models. Therefore, the model was trained on fewer characters as n increased and could achieve a higher accuracy.

VII. CONCLUSION

In our work, we were able to show that using a Recurrent Neural Network model to classify pen-based characters achieves comparable accuracy to that achieved by the standard k-Nearest Neighbors approach. However, an important design aspect we wanted to analyze was run time across the various methods. In Figure 10, we perform an analysis on the run time of the various methods by varying the number of letter segments sampled from each character. From this, we see that even though we found that the RNN was able to perform similarly to (and sometimes even better than) the k-NN approach, in practice an RNN is actually two orders of magnitude slower than the k-Nearest Neighbor approach. Analyses such as these allow for interesting design considerations when deciding on the best classifier to use in practice. Based on these results, we can support literature in their conclusion that the k-Nearest Neighbor approach might be a better classifier in practical application of temporal digital character recognition.

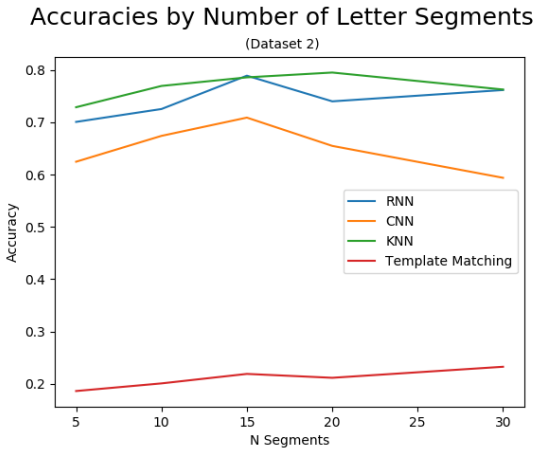

Fig. 9: Accuracy varying number of segments

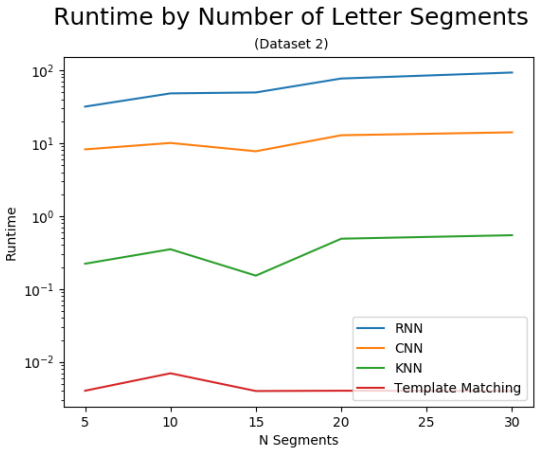

Fig. 10: Number of segments vs Runtime

VIII. INTERACTIVE APPLICATION

In order to collect and predict results using our own handwritten data, we created an interactive application that made use of the Tkinter module in Python. This application starts by prompting the user for the following information:

1) Collect data or Predict Data Mode
2) Number of samples to collect per letter
3) Name of user

Once the user inputs this information, an application is opened that allows a user to draw a letter using their mouse. After they complete each stroke, a bounding box is also drawn. At any given time, the user can press the $< Alt >$ key to determine the predicted letter based on what is currently drawn in the the GUI. Once they do this, a text of the top 3 predicted letters along with their corresponding confidences is displayed at top. Moreover, the user can press the $< Enter >$ key to clear the contents of the GUI at any given time the program is running.

If the user is in "collect data" mode, the letter that the user must draw in the GUI is indicated at the top. Moreover, for each stroke that the user draws for the indicated letter, that stroke is saved as a collection of (x, y) points in our data folder. The number of letters that the user inputs in **(2)** indicates how many of each letter to draw. For instance, if they input 2, then the order in which they draw letters is ["A", "A", "a", "a", "B", "B", "b", ..., "Z", "Z", "z", "z"].

If the user is in "predict data" mode, however, no letter is given to the user, and the user is free to draw whatever letter they like and predict at any time.

The program prompts for the user's name because, if the user is in "collect data" mode, we save the file names containing the points for each stroke with the following format: `{name}_{letter drawn}_{stroke number}_{hashid}.txt`, where hashid is a random 20-digit number used to differentiate between text files containing upper case and lower case labels.

The application was extremely useful in predicting results in real-time to see the progress of our work. Moreover, the application is a potential means of correcting the data

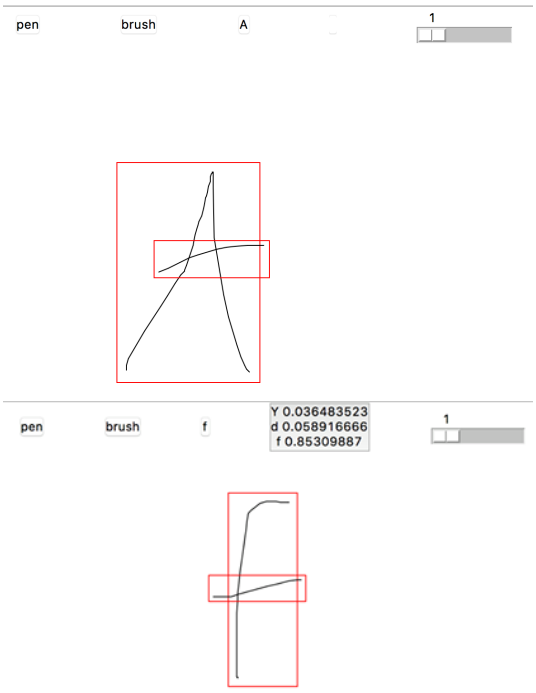

Fig. 11: ”Collect Data” Mode and ”Predict Data” Mode on Interactive Application

collection issue we mentioned regarding the relative sizes of letters — by introducing lines into the collection window we would be able to guide users to draw letters with correct scaling, thus making it possible to differentiate between C and c for example.

IX. FUTURE WORK

While we learned important lessons about online letter recognition, there were many questions and techniques unexplored by the end of our analysis which, given more time, we would have looked deeper into. One technique is using the Interactive Application to collect our own handwriting to train the model, as discussed earlier. We also would have tried to implement a different model for each number of strokes. In other words, implement a model for letters that require 1 stroke, another for letters that require 2 strokes, and so on. This would prevent us from having to concatenate the data points corresponding to different strokes for each letter and also give each classifier a smaller problem to solve. Finally, we could attempt to classify entire sentences at a time using bounding box segmentation, in which a user could write entire strings of letters and bounding boxes could be used to group strokes into letters, each of which could be classified independently. This in turn, if robust enough, could go into general purpose applications (such as note taking or information recording using a pen and tablet rather than typing).

CONTRIBUTION

We divided up the work for this project evenly among the members of the group. Each of the models and the baseline methods were written entirely by our team. The work was split up as follows:

- P. Karnati: Design RNN/CNN, Template Matching
- G. Madonna: Preprocessing of Dataset 1, Dataset 2; running experiments
- N. Prasad: Test RNN, k-NN algorithm, Interactive Application

ACKNOWLEDGMENT

We would like to thank the entire course staff of 6.867: Graduate Machine Learning at Massachusetts Institute of Technology for providing us with the knowledge, tools, and resources necessary for this research. Specifically, we would like to thank Professors Suvrit Sra, David Sontag, and Devavrat Shah as well as our mentor Flora Meng, for their support.

REFERENCES

[1] Connell & Jain. (1999). Template-based online character recognition. Pattern Recognition 34 (2001) 114.
[2] Llorens, David Prat, Federico Marzal, Andrs Vilar, Juan Castro, Mara Jos Amengual, Juan Barrachina Mir, Sergio Castellanos, Antonio Espaa Boquera, Salvador Gmez, Jon Gorbe-Moya, Jorge Gordo, Albert Palazn-Gonzlez, Vicente Ripolls, Guillermo Ramos-Garijo, Rafael Zamora-Martnez, Francisco. (2008). The UJIpenchars Database: a Pen-Based Database of Isolated Handwritten Characters.
[3] Prat, Federico Marzal, Andrs Martn, Sergio Ramos-Garijo, Rafael Castro, Mara Jos. (2009). A Template-based Recognition System for On-line Handwritten Characters.. J. Inf. Sci. Eng.. 25. 779-791.
[4] Alenius, Winblad, Sun. (2017). Handwriting Recognition: Artificial Intelligence Using Statistical Methods. Uppsala University Department of Information Technology.
[5] Williams. (2008) UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
[6] Llorens, Prat, Marzal, and Vilar. (2008) UJI Pen Characters Data Set. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
[7] UCI Machine Learning Repository. https://archive.ics.uci.edu /ml/datasets/UJI+Pen+Characters
[8] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.
[9] Chollet, F. (2015) keras, GitHub. https://github.com/fchollet/keras
[10] Williams, B., M.Toussaint, Storkey, A. (2006). Extracting motion primitives from natural handwriting data. Int. Conf. on Artificial Neural Networks (ICANN) (pp. 634643).
[11] J. Pradeep,E.Srinivasan,and S Himavathi,Neural network based handwritten character recognition system without feature extraction International Conference on Computer, Communication and Electrical Technology (ICCCET), pp.40-44,2011.
[12] Ch. N. Manisha, E. Sreenivasa Reddy, and Y. K. Sundara Krishna. (2016) Role of Offline Handwritten Character Recognition System in Various Applications. International Journal of Computer Applications 135 (2): 3033.