English Character Classification by Motion Sensing on Pen

A COGS118A Final Project

Chen Yang
Computer Science and Engineering
UC San Diego
chy099@ucsd.edu

Wanze Xie
Computer Science and Engineering
UC San Diego
waxie@ucsd.edu

ABSTRACT

Many technologies have been invented to transfer handwriting notes or scripts into digital form, including full-color touch screen and digital ink tablet. However, writing on screen or tablet feels quite different from writing on actual paper with a traditional pen, and there has not been a good solution to efficiently recognize people's handwriting on paper and convert them to real-time input to digital devices without image input. In this study, we built a machine learning pipeline to perform English character classification without image input, including a pen-attachable 9 degree of freedom motion sensing system. We deployed and tested four different supervised machine learning classifiers to this pipeline, including SVM, KNN, Random Forests, and AdaBoost. An important aspect of our study is to explore the feasibility of recognizing handwriting character based on unique handwriting strokes and to evaluate the performance of different learning methods on the pen motion data sets.

KEYWORDS

Handwriting motion capture, handwriting stroke, character recognition, motion recognition, KNN, Random Forest, SVM, AdaBoost.

1 INTRODUCTION

For a long time in history, pen and paper has been the dominant tools for people to record and keep information. Even though computers and keyboards has been introduced as another way of recording information, many people still preserve the habits of writing down things on the paper using traditional pen. With soaring development of technology, there have been many solutions to transferring handwriting notes or scripts to digital data in order to store handwritten information to digital devices for the convenience of reviewing and re-editing. There are currently two mainstream ways of achieving this goal. A popular way is digital pen like Surface Pen with Surface Pro and Apple Pen with iPad, or like writing pad for computers where users can directly write characters on the screens. Nevertheless, these solutions require expensive equipment and the feeling of writing is quite different from writing on the paper using traditional pen or pencil. On the other hand, hypertext reading (Rakefet, 2016) and writing (Subrahmanyam, 2013) can be distracting and often reduces people's focus on their work. Another solution is to use image-processing tools such as Optical Character Recognition (OCR) (Smith, 2007) to transfer handwritten scripts to digital characters in document files, but OCR technology require user to have an image version of their handwriting, which is not efficient.

Inspired by the above existing solutions, we are trying to implement a simpler and more intuitive way to combine both the advantage of writing on the paper and the convenience of keeping record of information on the digital device, so as to easily transfer the character the user is writing on the paper using traditional pen to digital and editable text document in real-time. By observation and user study, we discovered that when the user is writing down characters on the paper, the movement trajectory of the tail end of the pen is distinguishable and possibly unique. Therefore, this study is to explore the possibility and feasibility of recognizing the character the user is writing by keeping track of the movement of the tail end of a pen. In this study, we will be collecting the data of writing different characters, tuning different multi-class classifiers in machine learning to train our model and predicting the character the user is writing real time.

In this work, we primarily focus on training models to distinguish three basic English characters: a, b, and c. With our best efforts, we respectively collected approximately a thousand handwriting data for each character from different person as our raw data. We also 3D printed a small model for mounting a motion sensor onto a normal sharpie for handwriting data collecting and uses Arduino as transferring interface between sensor end and computer end. The goal for this prototype would be that when each new character is written, we will be able to recognize the pattern and print corresponding characters on the computer screen in the real-time.

2 METHOD

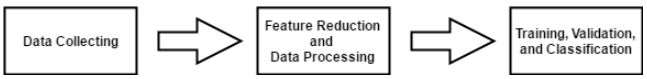

Figure 1: The machine learning pipeline we build for this study.

Figure 1 shows the pipeline we build for our study, which includes four different stages. The rest of this section will describe how we designed each of the step in the pipeline.

2.1 Data Collecting

In order to get the most accurate data sample using the sensor we can afford, we experimented on both MPU6050 6-axis motion sensor and MPU9250 9-axis motion sensor, and we compared the data divergence of using 6-axis motion sensor and 9-axis motion sensor. Table 1 shows the comparison between these two different sensors.

	MPU6050	MPU9250
Degree of Freedom	6	9
Built-in Sensors Type	Accelerometer Gyroscope	Accelerometer Gyroscope Magnetometer
Clock Interval	20-30ms	12-15ms

Table 1: Comparison between MPU6050 and MPU9250 Motion Sensor.

The most significant difference between these two sensors is that MPU6050 6-axis motion sensor generates 6 data point for each sample to report the current motion state of the sensor, and MPU9250 9-axis motion sensor introduces 3 more data point for each sample by detecting magnetic field.

To capture the motion of the tail end of a pen, we designed a 3D model and printed it out so that we can plug the end of a pen into the model, and attached the motion sensor to the top of a pen. Figure 2 shows the rendered 3D model, and Figure 3 shows the 3D printed version of the model we designed.

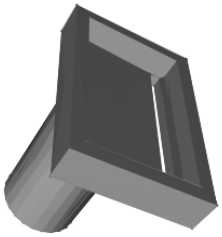

Figure 2: The rendered 3D model result of pen sensor mounting device. This model is the real model we used for 3D-printing.

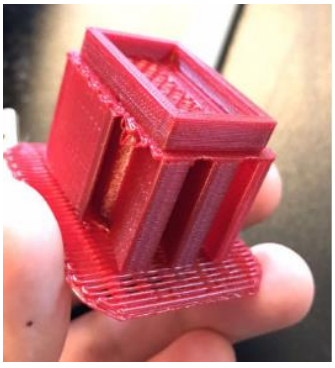

Figure 3: The printed result of our 3D model of the mounting device.

In order to make the hardware programming smooth for the study, we used Arduino Uno R3 as the hardware motherboard. We manually soldered motion sensor with the motherboard, and programmed the hardware source code for reading sensor's data and streaming processed data to computer. We also designed a control system to make the data more precise, which will be explained in the upcoming paragraph. What's more, a 2x16 pixel LED display is also programmed to the entire system, providing real-time feedback when collecting the data. Figure 4 shows a real image of the entire hardware system.

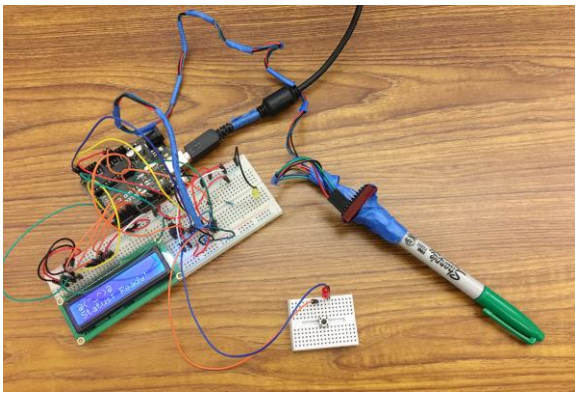

Figure 4: The hardware-sensing prototype we build for collecting and preprocessing pen's motion data.

One challenge during collecting data is that we only want to collect the motion of the pen when the pen holder is actually writing characters, and therefore the timing of the start and the end of writing a character can be hard to accurately detect based on the data variation itself. Therefore, we implemented an extra push down button for the sake of making collecting data easier. To collect the temporal data of the motion of the pen when writing each single character, the writer will press the push down button right before he or she is about to write a character, and release the push down button by the time he/she finished writing a character. Figure 5 shows a volunteer is working on creating data sample for this study.

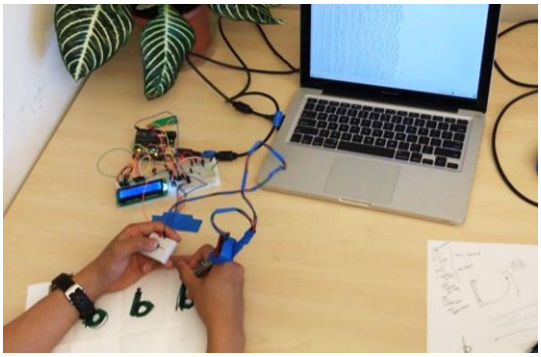

Figure 5: A volunteer is creating data sample for this study.

However, the fact is that some features will still lose and some noises will still be introduced because it is hard for writer to accurately press the push down button at exactly the time he or she started to write. What's more, the lasting time of writing a character can vary among different writers and even for same individual when writing different characters. To neutralize the noise and feature missing, we have figured out a way to normalize the data to extract out the features and this technique will be introduced in the data processing section.

2.2 Feature Reduction on 6/9 Degree of Freedom (DoF) Motion Sensor

MPU9250 motion sensor has 9 degree of freedom, which means each data sample contains 9 data points: accelerometer data on xyz axis, gyroscope data on xyz axis, magnetometer on xyz axis. In comparison, MPU6050 motion sensor has 6 degree of freedom, which means each data sample contains only 6 data points: accelerometer data on xyz axis, gyroscope data on xyz axis. From our user study, we discovered that people usually finish writing a character within 900-1200ms. In the best use case, these motion sensors are being able to return a motion sampling data in 15ms. This means, each character will contain around 60-80 data points. By multiplying number of data points and degree of freedom, we can estimate the number of feature will be around 550-800. Based on our homework experience, we realized that using feature size 550-800 will result as extremely long training and cross-validation time. So we decide to perform feature reduction, which can reduce the number of feature point for writing each character.

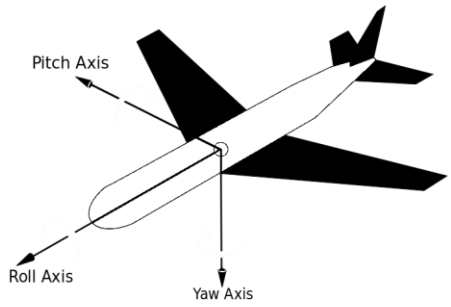

Figure 6: The definition standardized aircraft principal axes data: Yaw, Pitch, and Roll.

After researching different approach, we decided to convert the data collected by the sensor to standardized aircraft principal axes data: Yaw, Pitch and Roll. We performed this calculation on the Arduino motherboard, and use output stream to transfer calculated yaw pitch and roll data by using serial communication in the real-time to visualize the divergence of data. By calculating the yaw pitch roll based on data points returned by motion sensor, we achieve the feature reduction without down sampling the data, which keeps the data integrity.

Normally, calculating yaw, pitch, and roll only require the data from accelerometer and gyroscope. However, by adding magnetometer data to the calculation process, the yaw's value can be more stable and accurate. Base on this discover, we decided to use MPU9250 9-axis sensor because the variation of data is smoother and more stable with this sensor.

2.3 Data Processing

Each time when we recording the raw data, we performed yaw, pitch, and roll calculation on Arduino motherboard, then we stream outputted the yaw, pitch, and roll value combine with raw 9 DoF data to the serial monitor. Then on the computer, we save these data to a timestamped log file. Figure 7 shows an overview of this processing stream.

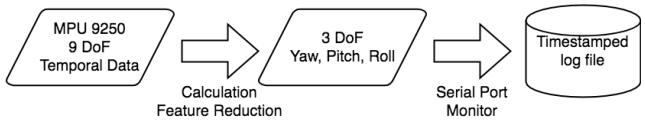

Figure 7: The flow chart showing the process of how data is read from MPU9250 and transfer to a log file for next step processing.

The sensing data from MPU9250 is temporal. Most of the machine learning classification model we learned in the class have fixed number of scalar features. In this case, we need to convert a series of temporal data to a row vector data that can be processed by using classifiers in scikit-learn library. Therefore, we decided to flatten all the data points that are collected in certain period for single character, and use them as features of this individual data.
The biggest challenge in this preprocessing pipeline is that the time it takes to write a single character varies a lot under different writer and cases, the amount of data collected for each character is different. Thus, we need to find a way to normalize the data so that the number of features for each data stays the same. Based on our implementation of MPU9250 motion sensor, our sensor will return the yaw pitch and roll data for every 12-15 milliseconds, and writing a single character takes around 900 milliseconds to 1200 milliseconds, so the first step for normalization is to extract out the pattern of the variance of the temporal data (yaw pitch and roll) during the single character writing. Then using interpolation, we can approximately predict the data during any time in this specific time period. In order to preserve as many meaningful features as possible without wasting too much computation power, we decided to up sampling and extract out 300 features for each data by retrieving 300 yaw pitch raw data from the interpolation model calculated from the raw data sample. At last, we iterate the same process for each data in our sample, Table 2 shows the data size after data processing.

Data Set	Sample Number	Feature Number
Character “a”	1,022	300
Character “b”	1,049	300
Character “c”	1,155	300

Table 2: Sample number and feature number of the raw input data set for each character.

In order to compare the difference between classifiers and data sets, we designed four different combinations of data set for performing training, cross-validation and testing. Table 3 shows different combination of the data set and their size.

Combination	Sample Number	Feature Number
“a-b”	2,071	300
“a-c”	2,177	300
“b-c”	2,204	300
“a-b-c”	3,226	300

Table 3: Combined data set and feature information.

2.4 Classification and Model Training

To train our model, we tested above “a-b” “a-c” “b-c” “a-b-c ”data sets using 4 different classifiers including: Support Vector Machines (SVM) with different kernels, K-Nearest Neighbour (KNN) with different algorithms and distance calculation, Random Forests with different way of different feature selector, and AdaBoost with different algorithms and estimation. For each classifier, we tuned different hyper-parameters in each function to find the one with the best testing error and best performance using techniques included data shuffling, cross validation and pruning. We will introduce and explain the experiment methodology in details in the next section.

A great challenge we met in the training was that the feature dimension is huge compared to the computation of our personal computer, and we hope to get the classification result for each classifier for all classification goals including classify “a” and “b”, classify “b” and “c”, classify “a” and “c” and classify “a”, “b”, and “c”. It takes very long time for us to complete even a single training process. Therefore, we not only optimized our algorithm using multiprocessing and multithreading in python, but also deployed our code to Google Cloud Compute Engine, where we established two 24-core instances for our model training. With this effort, we got our desired training result in reasonable amount of time, and we analysed the classification difficulty for different characters as well as between different handwritten characters, which we will be discussing in the conclusion section of this study. Figure 8 and Figure 9 shows the information and screenshot of the cloud instance we are using for this study.

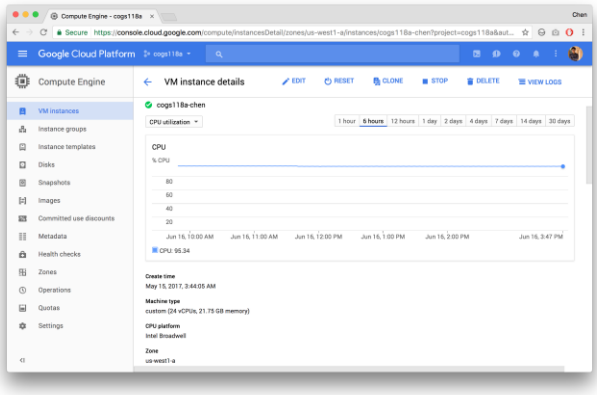

Figure 8: The Google Cloud Management Console showing the details of the instances we are using for training and cross-validation.

Figure 9: The System Monitor of Ubuntu LTS 16.04 showing the Python program is fully loaded on all 24 CPU cores that are available in the stack.

3 EXPERIMENT SETUP AND RESULTS

3.1 Support Vector Machine (SVM)

We implemented support vector machine (SVM) both on linear kernel and on RBF kernel to test out on our preprocessed dataset. In order to achieve the best performance, we tuned different hyper-parameter in the classifier. Table 4 shows the list of conditions, kernels, and the range of hyper-parameter we have tested in our SVM experiment.

Cond. ID	Kernel Type	C (Lo-Hi, St.)	Gamma (Lo-Hi, St.)	Cross-validation
1	Linear	(0.1-99.9, 0.2)	Auto	10
2	RBF	(0.1-99.9, 0.2)	Auto	10
3	RBF	Best C in Cond. 2	(0.001-1, 0,001)	10

Table 4: Experiment condition details for SVM experiment.

After deploying the SVM codebase to the Google Cloud, we performed training and cross-validation on four different data set combinations that we mentioned in the Section 2.3. However, although we are utilizing 24 CPUs on Google Cloud, training and cross-validation on "a-c", "b-c", and "a-b-c" takes extremely long time: after 24 hours of validation, the program only finish processing less than half of the data. So we decide to not performing SVM experiment on "a-c", "b-c", and "a-b-c". For "a-b" data set, we are able to finish the experiment in a reasonable amount of time, and this Table 5 shows the validation error, training error, and testing error under the best hyperparameters we gained from the experiment. Condition ID are shared with the setup table.

Cond. ID	Best Parameters	Validation Error	Training Error	Testing Error
1	C: 0.1	0.0857964	0.0144196	0.0687134
2	C: 49.9	0.4809033	0.4931506	0.5336257
3	C: 49.9 Gamma: 0.001	0.5336257	0	0.1885964

Table 5: Best parameters, validation error, training error, and testing error for SVM experiment on "a-b" dataset.

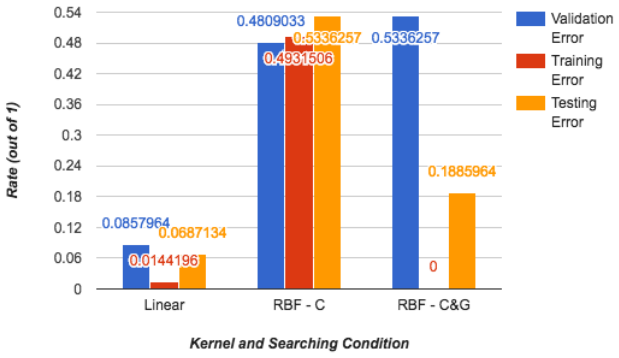

Figure 10: Validation, Training, Testing Error comparison between three SVM experiment condition.

Based on the Table 5 and Figure 10 shows above, we can found that linear kernel performed much better than RBF kernel on validation error, training error, and testing error. Testing error on RBF kernel is really high, which shows that the model is not useable in the real case. To verify the correctness of our experiment, we plotted out the relationship between C's value and validation error.

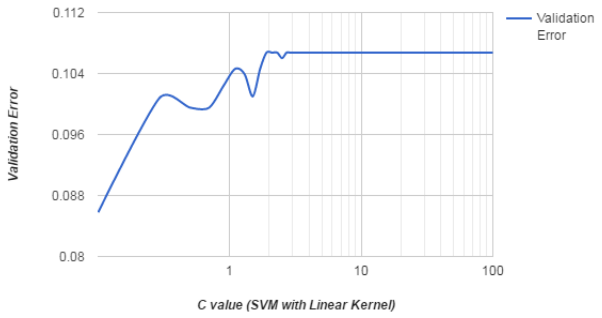

Figure 11: C's value verse validation on Linear kernel SVM.

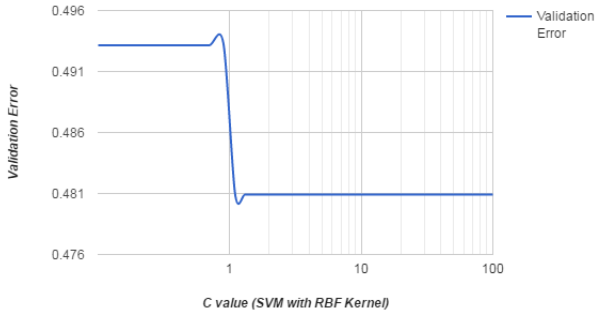

Figure 12: C's value verse validation on RBF kernel SVM.

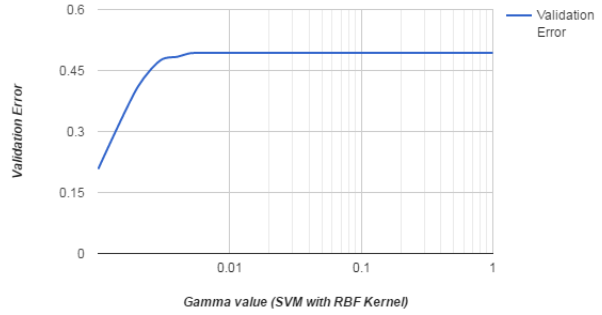

Figure 13: Gamma's value verse validation on RBF kernel SVM.

Based on the Figure 11-13, we can see that on both linear and RBF kernels, after increase C and gamma to certain values, the validation error stays the same. This means the searching range in our program is large and detailed enough to cover all necessary

range of hyperparameters in order to minimalize the validation error.

3.2 K-Nearest Neighbors (KNN)

We implemented K-Nearest Neighbors (KNN) both by using Ball Tree and K-d Tree algorithm. On top of that, we used five different way of calculating distance metrics: Minkowski, Euclidean, L1, L2, and Manhattan. In each cases, we tuned the number of nearest neighbors when KNN is producing prediction. Table 6 shows the list of conditions, algorithm, distance metric, and the range of hyperparameters we have tested in our KNN experiment.

Cond. ID	Algorithm Type	Distance Metric	Number of Neighbors	Cross-validation
1	Ball Tree	Minkowski	1 to 100	10
2	Ball Tree	Euclidean	1 to 100	10
3	Ball Tree	L1	1 to 100	10
4	Ball Tree	L2	1 to 100	10
5	Ball Tree	Manhattan	1 to 100	10
6	K-d Tree	Minkowski	1 to 100	10
7	K-d Tree	Euclidean	1 to 100	10
8	K-d Tree	L1	1 to 100	10
9	K-d Tree	L2	1 to 100	10
10	K-d Tree	Manhattan	1 to 100	10

Table 6: Experiment condition details for KNN experiment.

Cond. ID	Best Tree Counter	Validation Error	Training Error	Testing Error
1	1	0.0129710	0	0.0043859
2	1	0.0129710	0	0.0043859
3	1	0.0079192	0	0.0029239
4	1	0.0129710	0	0.0043859
5	1	0.0079192	0	0.0029239
6	1	0.0129710	0	0.0043859
7	1	0.0129710	0	0.0043859
8	1	0.0079192	0	0.0029239
9	1	0.0129710	0	0.0043859
10	1	0.0079192	0	0.0029239

Table 7: Best parameters, validation error, training error, and testing error for KNN experiment on "a-b" dataset.

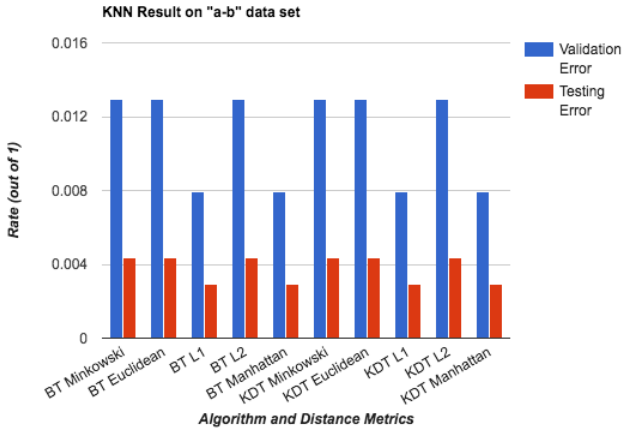

Figure 14: Experiment result comparison of different algorithm and distance metrics with "a-b" dataset.

Table 7 and Figure 14 shows the KNN experiment result we performed on "a-b" data set. We can see that L1 and Manhattan distance metrics performed relatively better than other three distance metrics. There is no difference on the validation, training and testing error between two different algorithms: K-d Tree and Ball Tree. The only difference is the run-time speed.

Cond. ID	Best Tree Counter	Validation Error	Training Error	Testing Error
1	1	0.0164765	0	0.0139082
2	1	0.0164765	0	0.0139082
3	1	0.0123527	0	0.0125173
4	1	0.0164765	0	0.0139082
5	1	0.0123527	0	0.0125173
6	1	0.0164765	0	0.0139082
7	1	0.0164765	0	0.0139082
8	1	0.0123527	0	0.0125173
9	1	0.0164765	0	0.0139082
10	1	0.0123527	0	0.0125173

Table 8: Best parameters, validation error, training error, and testing error for KNN experiment on "a-c" dataset.

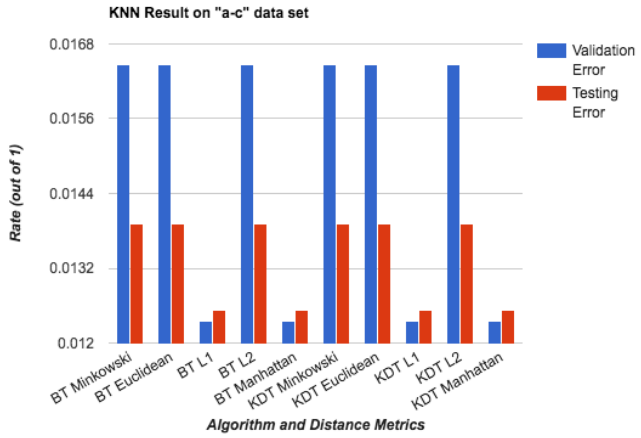

Figure 15: Experiment result comparison of different algorithm and distance metrics with "a-c" dataset.

Table 8 and Figure 15 shows the KNN experiment result we performed on "a-c" data set. Similar to the "a-b" data set, when we perform KNN experiment on the "a-c" data set, L1 and Manhattan distance metrics still performed better than other three distance metrics.

Cond. ID	Best Tree Counter	Validation Error	Training Error	Testing Error
1	1	0.0162437	0	0.0151098
2	1	0.0162437	0	0.0151098
3	1	0.0108291	0	0.0164835
4	1	0.0162437	0	0.0151098
5	1	0.0108291	0	0.0164835
6	1	0.0162437	0	0.0151098
7	1	0.0162437	0	0.0151098
8	1	0.0108291	0	0.0164835
9	1	0.0162437	0	0.0151098
10	1	0.0108291	0	0.0164835

Table 9: Best parameters, validation error, training error, and testing error for KNN experiment on "b-c" dataset.

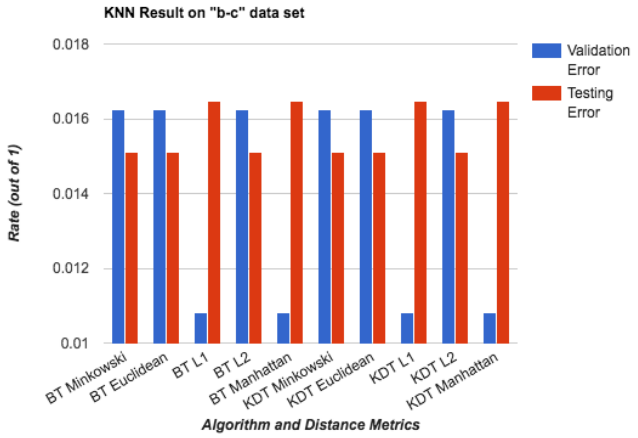

Figure 16: Experiment result comparison of different algorithm and distance metrics with "b-c" dataset.

Table 9 and Figure 16 shows the KNN experiment result we performed on "b-c" data set. When we perform KNN experiment on the "b-c" data set, L1 and Manhattan distance metrics did not perform better than L2, Minkowski, and Euclidean distance metrics. However, when we look at the validation error, the L1 and Manhattan distance metrics actually have really low validation error. We guess the reason for that is the KNN already overfitted the data.

Cond. ID	Best Tree Counter	Validation Error	Training Error	Testing Error
1	1	0.0198905	0	0.0225352
2	1	0.0198905	0	0.0225352
3	1	0.0161910	0	0.0178403
4	1	0.0198905	0	0.0225352
5	1	0.0161910	0	0.0178403
6	1	0.0198905	0	0.0225352
7	1	0.0198905	0	0.0225352
8	1	0.0161910	0	0.0178403
9	1	0.0198905	0	0.0225352
10	1	0.0161910	0	0.0178403

Table 10: Best parameters, validation error, training error, and testing error for KNN experiment on "a-b-c" dataset.

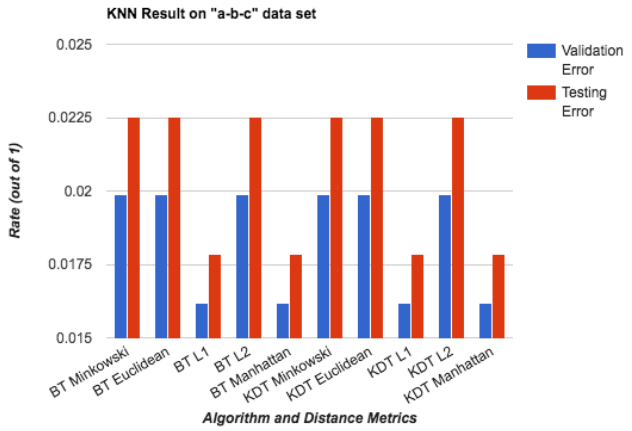

Figure 17: Experiment result comparison of different algorithm and distance metrics with "a-b-c" dataset.

Table 10 and Figure 17 shows the KNN experiment result we performed on "a-b-c" data set. Situation is similar to the "a-b" and "a-c" data set, which L1 and Manhattan distance performed better than the other three distance metrics.

3.3 Random Forest

We tested the performance of Random Forest classifier on our dataset. To achieve the best result, we tuned the number of feature to consider when looking for the best split on: sqrt(n_features), log2(n_features), and n_features. We also tuned the number of trees we build when we create the forest. In our first round of testing, we try to search the best validation error between the 10 to 128 trees. Table 11 shows the list of conditions, and max number of features to consider in our random forest classification experiment.

Cond. ID	Max Feature Selector	Number of Tree (Lo-Hi, Step)
1	sqrt(n_feature)	(10-128, 1)
2	log2(n_feature)	(10-128, 1)
3	(n_feature)	(10-128, 1)

Table 11: Experiment condition details for Random Forest experiment (first round).

After the first round, when we are running the classification model for character a and b, we found that the more trees we have in the forest, the lower the validation error it is, which means the model might have stronger generalization ability when this metrics is larger. This conclusion also applies to all other data sets ("a-b", "a-c", "b-c", "a-b-c") and feature selectors ("sqrt", "log2", "None"). Figure 18-20 shows the trend of different feature selectors on the "a-b" data set.

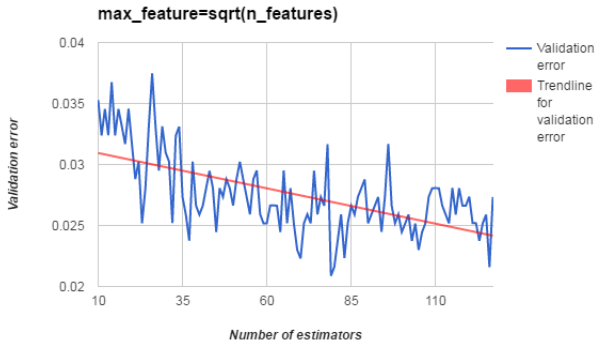

Figure 18: Number of estimators verse (10-128) validation error under sqrt feature selector.

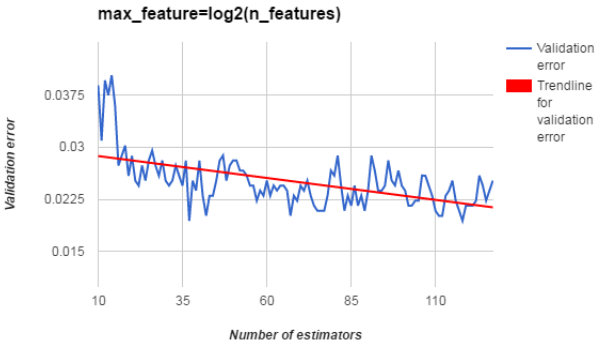

Figure 19: Number of estimators (10-128) verse validation error under log2 feature selector.

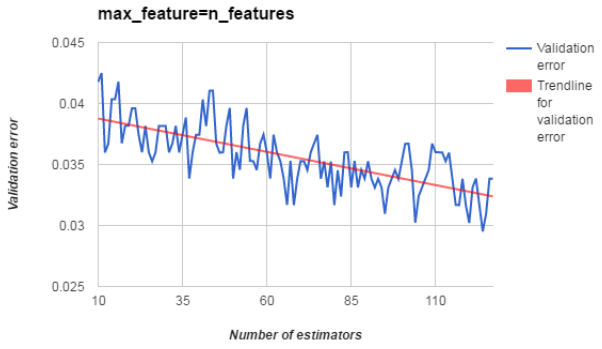

Figure 20: Number of estimators (10-128) verse validation error under without limit number of features.

Based on Figure 18-20, this phenomenon lead to us to increase the number of trees in the forest and train the model again. In the second round, we increased the range of testing from 10 to 128 trees to 10 to 2048 trees, trying to figure out in what range of the

number of estimators (trees) would produce the best generalization ability. Table 12 shows the new conditions we designed for random forest experiment.

Cond. ID	Max Feature Selector	Number of Tree (Lo-Hi, Step)
1	sqrt(n_feature)	(10-2048, 7)
2	log2(n_feature)	(10-2048, 7)
3	(n_feature)	(10-2048, 7)

Table 12: Experiment condition details for Random Forest experiment (second round).

Interestingly, according to the new graph in Figure 21-23, we discovered that from the first 400 of point sample, the validation error drops dramatically as the number of estimators (trees) increase, but after the number of estimators increases over 400, the validation error fluctuate up and down around certain value but no longer show the pattern of decreasing. Figure 21-23 shows the change of validation error when increasing the number of estimators.

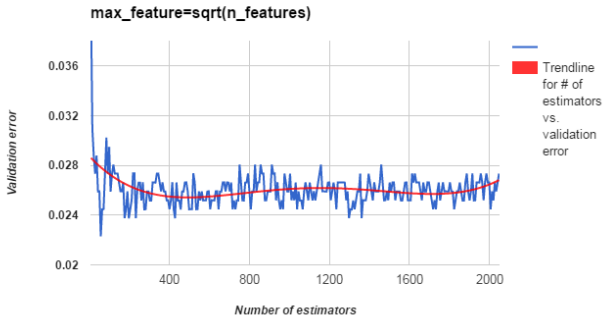

Figure 21: Number of estimators verse validation error under sqrt feature selector.

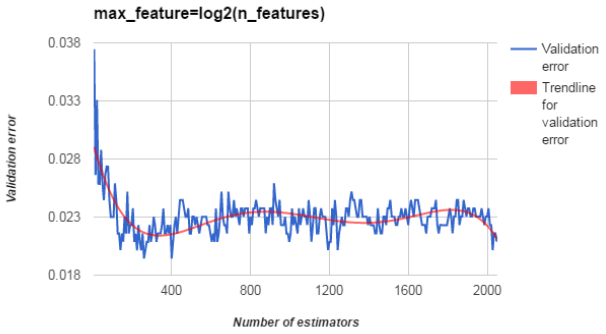

Figure 22: Number of estimators verse validation error under log2 feature selector.

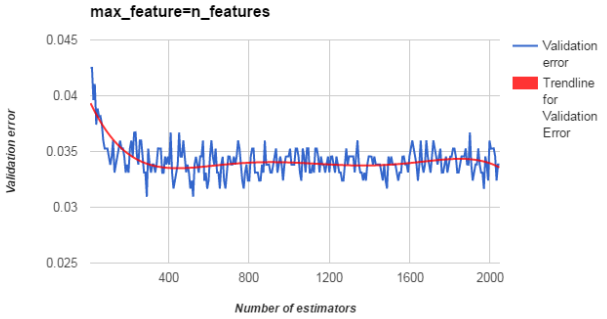

Figure 23: Number of estimators verse validation error without limit number of features.

The training result for data set "a-b" verifies our observation and as shown in Table 13, the log2 feature selector turned out to be the best feature selector for data set "a-b". With this finding, we keep exploring how different feature selector influence the validation error, training error, and testing error. Table 13 summarize the result we get from random forest experiment.

Cond. ID	Best Tree Number	Validation Error	Training Error	Testing Error
1	146	0.0259107	0	0.0190058
2	377	0.0230277	0	0.0116959
3	451	0.0316508	0	0.0438596

Table 13: Best parameters, validation error, training error, and testing error for Random Forest experiment on "a-b" dataset.

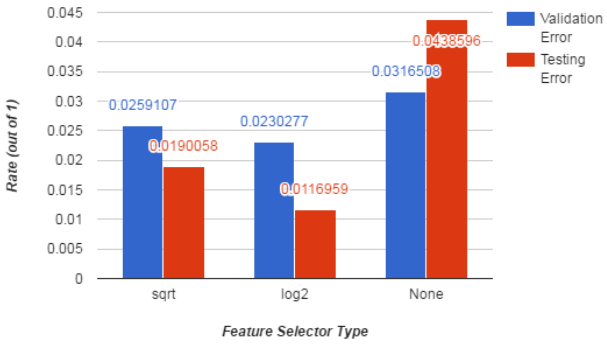

Figure 23: Comparison between different types of feature selectors in Random Forest experiment.

Based on Figure 23, by comparing different type of feature selectors, we can clearly see that log2 feature selector has the lowest training and testing error among other two types of feature selectors. Moreover, since the log2 feature selector reduced the

feature when we splitting the tree, the runtime for log2 is also the fastest among these three options. Table 14 shows the comparison between different runtime on these three feature selectors.

Feature Selector Type	Runtime (Minutes)
sqrt(n_features)	48
log2(n_features)	34
None	679

Table 14: Runtime comparison between different feature selectors on the Random Forest experiment.

Based on Table 14, we can see that with log2 feature selector, random forest performed the best with faster runtime and better generalization and testing error.

3.4 AdaBoost

At last, we trained our model using Adaboost algorithm model based on decision tree classifier. As argued by Mease (Mease, 2008), we learned that it takes a long time for AdaBoost to run until it converges, and 1000 iterations is a suggested number for AdaBoost, so we hoped to verify this result by cross validate the "n_estimators", which is the number of iterations for AdaBoost to finish. We will also cross validate the tree depth {1,2,3,4} for the decision tree classifier to see which model will produce the best performance. Table 15 summarize the experiment conditions we designed in AdaBoost experiment:

Cond. ID	Algorithm	Tree Depth	Number of Estimators (Lo-Hi, Step)
1	SAMME	1	Grid (100-2000, 100) Binary (x10 times)
2	SAMME	2	Grid (100-2000, 100) Binary (x10 times)
3	SAMME	3	Grid (100-2000, 100) Binary (x10 times)
4	SAMME	4	Grid (100-2000, 100) Binary (x10 times)
5	SAMME Real	1	Grid (100-2000, 100) Binary (x10 times)
6	SAMME Real	2	Grid (100-2000, 100) Binary (x10 times)
7	SAMME Real	3	Grid (100-2000, 100) Binary (x10 times)
8	SAMME Real	4	Grid (100-2000, 100) Binary (x10 times)

Table 15: Experiment condition details AdaBoost experiment.

Since dataset "a-b" is the most typical pair, we started with classification for dataset "a-b". Figure 24 shows the validation error and testing error of the AdaBoost model under different depth for the decision tree classifier under best-tuned number of iterations. It turned out that when the depth of the decision tree gets larger than 2, it is already good enough for the training model and will have little effect to testing error and validation error.

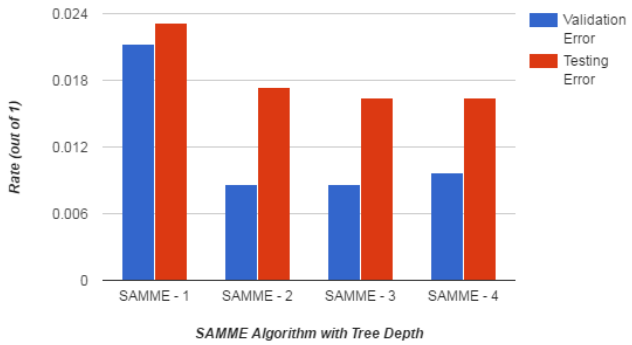

Figure 24: Comparison between different depths of tree depth on SEMME algorithm in AdaBoost experiment.

Figure 25 shows the relationship between validation error and the number of iterations. As shown in the trend line in the graph, the validation error has a general trend of decreasing as the number of iterations get large, but as mentioned by Mease, 1000-iteration is already good enough for AdaBoost to generalize the data model. We also integrated the trend line of the validation error pattern across different depth of decision trees. Again, it verifies our conclusion that when the depth is larger than or equal to two, it is already good enough for the training model.

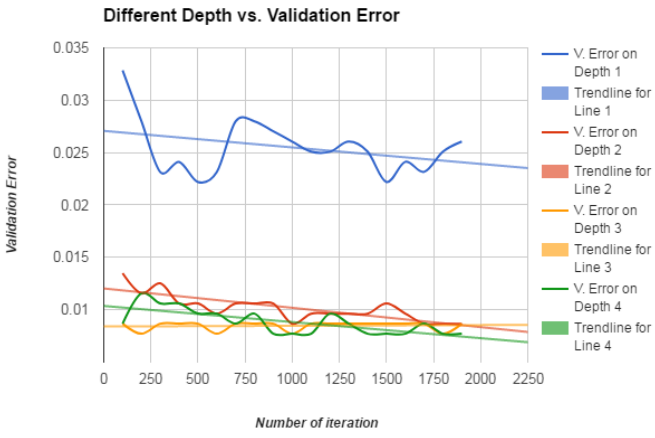

Figure 25: Comparison between different depths of tree depth on SEMME algorithm in AdaBoost experiment in terms of validation error verse number of iteration.

When we are testing with the two algorithms {SAMME, SAMME_R}, it turned out that SAMME_R in general has a better performance than SAMME because it takes less time to run, and shows stronger generalization ability even when the number of iterations is small. An interesting thing to note is that the training

error under best tuned hyperparameters are 0, but we do find training error when the number of iterations is very small, it might because that when the number of iterations get large, there will be little training error rate, and even when the training error reaches zero, the number of iterations will continue have effect on the validation error and testing error. We also used the model to train dataset b-c, a-c and a-b-c, and they all show the trend that agrees with the result shown above, but the resultant accuracy is not as good as other models like KNN and Random Forests.

3.5 Cross-Model Performance Comparison

Model Type	Validation Error	Training Error	Testing Error
SVM	0.0857964	0.0144196	0.0687134
KNN	0.0079192	0	0.0029239
Random Forest	0.0230277	0	0.0116959
AdaBoosting	0.0096340	0	0.0164092

Table 16: Experiment result on cross-model comparison.

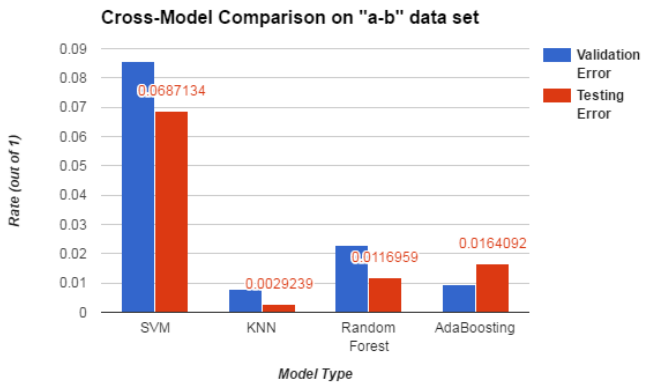

Figure 26: Cross-Model comparison on the performance on "a-b" dataset.

Based on Table 16 and Figure 26, across different models, it turned out that K-Nearest-Neighbors (KNN) model has the best performance. It takes the least time to run and gives the best accuracy among all the training models, and when running on dataset a-b, it has accuracy as high as 99.71%, with testing error only 0.0029 when using l1 and Manhattan as distance metrics. KNN also shows strong classification ability on data set a-c and b-c, and this is why we pick this training model to train on the dataset a-b-c for multi-class classification, and it does give a decent accuracy as high as 98.2%. Other model also shows good performance on training accuracy except for SVM with RBF kernel. The integrated validation, training, and testing error of all the models on "a-b" data set are in the table above.

4 CONCLUSIONS

This paper proposed a potential way of recognizing people's handwriting based on the stroke of handwriting instead of using image recognition (such as OCR), which enables the possibility of real-time converting of handwriting information to digital device by the time user is writing on the traditional paper using a regular pen. Our study designed and developed the pipeline starting from collecting data using pen's motion sensor and preprocessing temporal data to static data, to cross validating different hyperparameters on different training models and generate classification result on simple English characters.

We enriched our classification result by testing on different pair of datasets so as to explore the how various pen's motion and stroke pattern might affect the classification difficulty. We started by analyze set "a-b" and "a-c", and it turned out that in most models except AdaBoost, the pair "a-b" often produce a better training result than "a-c". We believe it is because that if the stroke of a character is part of the stroke of another character, or if they have a similar stroke then it might be harder for the model to learn. However, it does not mean that they cannot be classified, since in the best training result, the testing error can still be as low as around 1.5% in the best training model. In general, KNN model shows strongest ability and best performance to classify our entire datasets ("a-b", "a-c", "b-c"), which can also produce highly accurate even for multi-class classification ("a-b-c").

We believe we could achieve better results even when classifying more characters since we wasn't able to adopted method of temporal data classification (Tseng, 2008) due to limitation of the knowledge of this class. As an outlook for our study, with advanced algorithm such as RNN, auto-correction, and natural language processing, it might be possible to achieve converting handwriting notes to computer devices or cloud system in real-time, based on the classification of pen's motion.

5 BOUNUS POINTS

5.1 Novel Ideas and Applications

a) We did many researches on people's behavior of handwriting and did competitive analysis of many existing solutions, and finally came up with this innovative way by to just use handwriting stroke as an input to computer device without the need for writing pad or touch screen.

b) The conclusion above has shown that this idea is feasible for at least small amount of characters, and it could be commercializable if more advanced learning method is adopted.

5.2 Large Efforts on Our Own Data Collection, Preparation, and Preprocessing

a) We tested on different motion sensors to find the best sensor for data recording.

b) We built an actual physical prototype using Arduino UNO R3 kit and traditional sharpie for data collecting as shown in the figure x.
c) We designed algorithm for feature deduction and algorithm to convert temporal raw data to 300 features for each English character for the convenience of classification analysis.
d) We deployed our code to Google Cloud Service in order to get more accurate training result with a better computing power.
e) We optimized our algorithm using multi-processing when running cross validation with different hyperparameters.

5.3 Comprehensive and State-of-the-art Classification methodology and Result.

a) For each training model, we tested them on all possible combination of data sets including "a-b", "a-c", "b-c" and "a-b-c", in an effort to research on how the stroke of different characters might affect the classification result.
b) The models we used include SVMs, KNN, Random Forests, and AdaBoost, and our classification result analyzed the performance of each model on our datasets.
c) We built up the whole pipeline from data collecting to classification result from scratch.

6 DOWNLOADS

1. Download pre-processing source code: http://azureric.org/static/cogs118a/final/pre_processing.py
2. Download experiment source code:
    a. SVM: http://azureric.org/static/cogs118a/final/train_svm.py
    b. KNN: http://azureric.org/static/cogs118a/final/train_knn.py
    c. Random Forest: http://azureric.org/static/cogs118a/final/train_rf.py
    d. Ada Boosting: http://azureric.org/static/cogs118a/final/train_boosting.py
3. Download raw data set:
    a. Character "a": http://azureric.org/static/cogs118a/final/run_letter_a.csv
    b. Character "b": http://azureric.org/static/cogs118a/final/run_letter_b.csv
    c. Character "c": http://azureric.org/static/cogs118a/final/run_letter_c.csv

ACKNOWLEDGMENTS

During this study, we have been fortunate to have the support and friendship of many individuals. First and foremost, we thank our professor, Zhuowen Tu. Over the weeks, Zhuowen has provided lots of great advises on our study, and he's suggestions have great inspired us to explore the new application of state-of-the-art machine learning field. We would like to next say thank you to all the TAs of this class: Long Jin, Zeyu Chen, and Daniel Maryanovsky. They have provided great support to our study and even helped us to create more data sample as our training and testing set.

REFERENCES

[1] Ackerman, Rakefet and Goldsmith, Morris. 2011. "Metacognitive regulation of text learning: On screen versus on paper." Journal of Experimental Psychology (Jan. 2007), 18-32. DOI: http://ucelinks.cdlib.org/10.1037/a0022086
[2] Subrahmanyam, Kaveri and Michikyan, Minas. 2013. "Learning from Paper, Learning from Screens: Impact of Screen Reading and Multitasking Conditions on Reading and Writing among College Students." IJCBPL (Apr. 2013). DOI: https://www.igi-global.com/10.4018/ijcbpl.2013100101
[3] R. Smith. 2011."An Overview of the Tesseract OCR Engine," Ninth International Conference on Document Analysis and Recognition (ICDAR 2007), Parana, 2007, pp. 629-633. DOI: http://ieeexplore.ieee.org/ 10.1109/ICDAR.2007.4376991
[4] Mease, David and Wyner, Abraham. 2008. "Evidence Contrary to the Statistical View of Boosting" Journal of Machine Learning Research 9 (Feb. 2008). DOI: https://jmlr.org/ N/A
[5] Tseng, Vincent S. and Lee, Chao-Hui. 2008. "Effective Temporal Data Classification by Integrating Sequential Pattern Mining and Probabilistic Induction" ScienceDirect (Oct. 2008), 1-30. DOI: https://doi.org/10.1016/j.eswa.2008.10.077
[6] Caruana, Rich and Niculescu-Mizil, Alexandru. 2006. "An Emprical Comparison of Supervised Learning Algorithms" ICML '06 (Feb. 2008). 161-168 DOI: https://http://dl.acm.org/10.1145/1143844.1143865