The Learning Keyboard

Using the Xbox Kinect to Learn User Typing Behavior

Jonathan Ellithorpe
jdellit@stanford.edu

Pearl Tan
pearltan@stanford.edu

***Abstract*—The Learning Keyboard is a machine learning system designed to guess what a user is typing solely by observing their hand movements on a keyboard. The system trains on a per-user basis using supervised learning, and generates feature vectors on a per-word basis. We show that on datasets consisting of more than 100 unique words the Learning Keyboard is able to guess words with over 80% accuracy.**

I. Introduction

Since the invention of the typewriter in the 1860s, keyboards have been and remain to be the primary method of character input for computers and various other devices. At an abstract level, the keyboard is only a method to translate actuation of the hands into a sequence of characters to be input to the computer device. In light of this, and with modern computer vision technology, we asked ourselves if the presence of a physical keyboard was even necessary to achieve this very basic goal. As a first step towards answering this question we designed the Learning Keyboard system, whereby the computer actually learns what the user is typing by observing only the movement of the user's hands. While the presence of a physical keyboard is required for training, once trained the keyboard is no longer needed for typing.

In this paper we describe our first efforts towards this goal, including the setup of a data collection station for recording a user's typing and hand motions, the extraction of different feature vectors from the data, and our performance results using a multi-class classification support vector machine.

II. Data Collection

For collecting the needed data for training we setup a monitoring station pictured in Figure 1. Suspended directly above the laptop keyboard at a distance of roughly 70 centimeters is a Kinect sensor equipped with an infrared camera, infrared projector, and RGB camera. The desk space below is marked off for consistent placement of the laptop with respect to the camera above.

To pull the image data from the Kinect sensor we used the open source OpenKinect driver for Mac [1]. The software comes with a program "record" which dumps all video data from the device to disk with a Unix time-of-day timestamp. In the recorded depthmap images, each pixel value denotes the distance of the object at that point in the image from the camera. Figure 2 shows what the camera captures from its viewpoint.

To capture key press information we wrote a small key-logger using the TKinter library [2]. We log every keypress made by the user and tag it along with a Unix time-of-day timestamp, which allows us to later sync up with the Kinect video data.

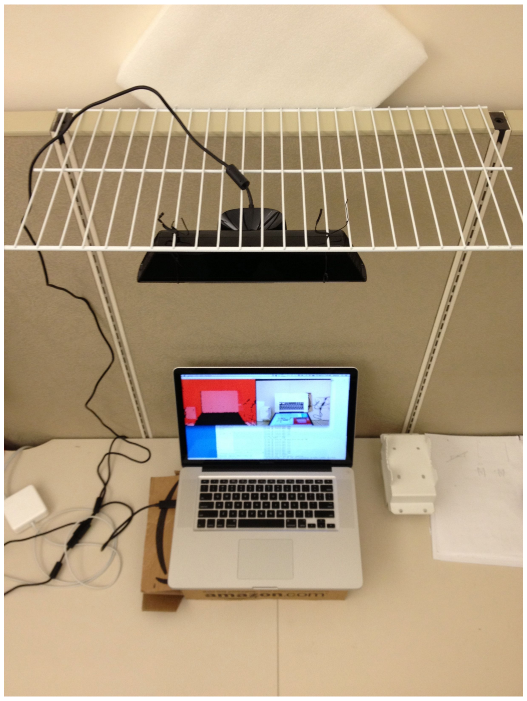


Fig. 1: Data capture station

Finally to simplify data processing and feature vector extraction, for each data capture session we author a dictionary file pre-defining the words to look for and extract out of the data. In the future it would of course be possible to automatically generate this dictionary file from the keylogging data itself, but for now we use this method to manually control which words are observed and learned on in the datasets.

III. Feature Vector Extraction

A. *Learning on Words, not Letters*

For our first implementation of the system we decided to extract feature vectors on a per word basis and not on a per key basis, as might seem at first obvious to try. This decision was the result of two simple observations. The first was that single key presses along the homerow of the keyboard (asdfjkl) were indistinguishable from each other given the resolution of

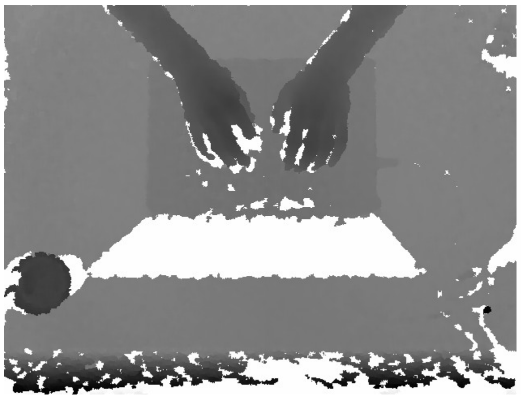


Fig. 2: Raw depth map capture from camera

depthmap data collected by the Kinect sensor (indeed, a single keypress is a mere 2-3 millimeters of movement). Therefore, learning would need to be done based on the whole position of the hands. The second observation was that the position of the hands during the press of a key is totally dependent on the word being typed anyway, resulting in any given key having as many feature vector "signatures" as there are words that include that key. While learning on a per key basis may nontheless still be worth a try, it is left as future work.

B. Image Data Preprocessing

Before extracting feature vectors for words from the collected image data, we first run the images through a proprocessor. The preprocessor removes the background from the image apart from the hands themselves (using a simple threshold filter), and crops the image to include only the area covered by the keyboard. Figure 3 shows an example of a preprocessed image.

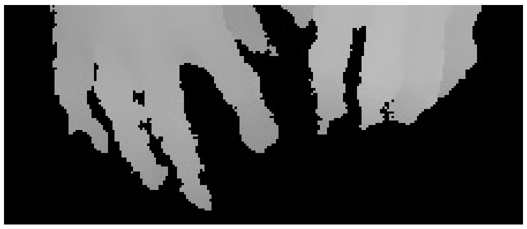


Fig. 3: Preprocessed depth map

C. Extraction Mechanism

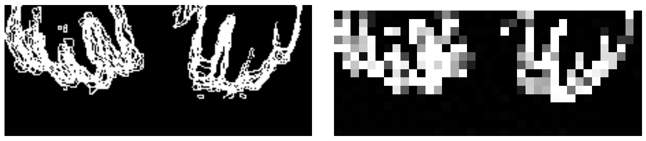
To build a feature vector for the occurence of a typed word, we first find the start and end times for the typing of that word in the keylogging data, and then use those times to splice out the recording of that word in the preprocessed depth map video stream. The result is a set of frames for the word that are then processed to generate a set of features describing the hand motions captured in the frames. In the next section we describe the three different types of feature vectors that we designed, and explore their performance in a support vector machine in the Results section.

IV. Feature Vectors

In this section we describe the types of feature vectors that we developed and tested.

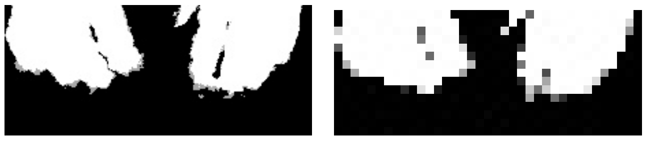
A. Edge Detector Type

For this type of feature vector we used the Canny [3] edge detector as implemented in the OpenCV library to turn the depth map data for a given frame of a given word into an edge map. The edge maps for all frames are then added together to produce a single frame for that word. See Figure 4 for an example. Our reasoning for using such a feature vector was that the superimposition of edge maps results in an image that seems to capture the motion of the hands during the typing of a word.



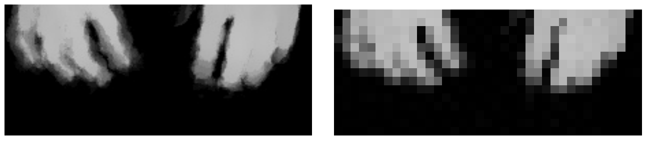
(a) No Downsampling (b) Downsampled

Fig. 4: Edge Detector Type Feature Vector



(a) No Downsampling (b) Downsampled

Fig. 5: Sum Type Feature Vector



(a) No Downsampling (b) Downsampled

Fig. 6: Average Type Feature Vector

The downside of this approach, however, was that since the resulting image in our implementation of the system was too large (225x95) to use in an SVM (to allow the computation to run fast enough), downsampling was required. Figure 4b shows the result of downsampling, and we can see that the image no longer seems to capture the "movement" of the hands. In the section on Results we see how downsampling affects the performance of an SVM on this type of feature vector.

B. Sum Type

This feature vector is much simpler than the Edge Detector type and just adds up the images from the frames captured for a given word. Since the background of all the depthmap images is black after preprocessing, and since the image is 8-bit greyscale, adding the images together results in an image that is bright white wherever the hands had moved to, producing a type of "snow angel" effect. Our logic in using this style of feature vector was therefore that the SVM may be able to use such "area-coverage" of the hands during the typing of a word as a clear signature of what word was typed. See Figure 5 for an example.

C. Average Type

This feature vector is the same as the sum type feature vector except that we take an average at the last step. Taking an average has the effect of revealing the amount of time the fingers have spent on various parts of the keyboard, which is lost in the sum type feature vector due to greyscale images having a 255 pixel value cap. See Figure 6 for an example of this type of feature vector.

V. Learning Model

Since the problem at hand was fundamentally a supervised learning problem with multi-class classifcation on words, we decided to use an off-the-shelf support vector machine called LIBSVM [4]. We chose default parameter settings, with the exception of using a linear kernel instead of the default radial kernel. All data in the Results section was generated using LIBSVM version 3.14 with these settings.

VI. Results

A. Datasets

To evaluate the performance of our feature vector types we captured four datasets described below, in order of increasing complexity, on which to test them.

1) "Plumpy Databases" Dataset: Our first dataset consisted only of the words "plumpy" and "databases". These words are typed solely by the right and left hands respectively (and so are easily distinguishable in the feature space).

2) "Charlie Samson" Dataset: Our second dataset also consisted of only two words, "charlie" and "samson", but this time each requires both hands to type (making them harder to tell apart in the feature space).

3) "Jumping Fox" Dataset: Our third dataset consisted of the 8 unique word sentence "the quick brown fox jumps over the lazy dog", and covers every letter of the alphabet.

4) "Night Before Christmas" Dataset: Twas was the largest dataset that we collected, and consists of typing the first 4 stanzas of "The Night Before Christmas" poem by Clement Clark Moore. These stanzas contain a total of 114 unique words (166 words in all).

B. Comparing Performance

To compare the performance of our different feature vector types, for each type and for each dataset above we generated a training set consisting of 10 samples for every unique word in the dataset. We then fed these training examples to the SVM for leave-one-out cross validation and recorded the results, shown in Figure 7. Note that for computational tractability we used a downsampling factor of 6 on our images to reduce the length of the resulting feature vectors from 21,375 to 555.

The "Plumpy Databases" two word dataset turned out to pose no challenge to any of the three feature vector types. The single handed typing of the two words came through clearly for each, all of which scored a perfect accuracy. When increasing the degree of complexity with "multi-hand-typed" words in the "Charlie Samson" dataset, however, we see performance begin to drop. Surprisingly, though, this reduced accuracy is held at 95% even when we increase the number of unique words to 8 in the "Jumping Fox" dataset, despite the fact that we are maintaining our 10 samples / word restriction for all datasets. We notice that the Edge Detector type feature vector begins to suffer in performance, however, and takes an even bigger hit when increasing the unique word count to 114 in the "Night Before Christmas" dataset. It's here that the different feature vector types' strength as a signal for a given word starts to become apparent. The Sum type shows itself to be an even better signature than the Edge Detector type, while the Average type leads the ranks, validating its preservation of time information as a useful quality and strong indicator of which word was typed.

The floundering Edge Detector feature vector was a curiousity to us since we had assumed from the start that it would be the best feature set. Our first suspicion was that downsampling may have been taking a greater toll on its performance compared to the other feature vector types, but this question was put to rest when we observed the performance of each type with zero downsampling (shown in Figure 8). Though we still don't know the reason for its lagging performance, perhaps variability in the resulting superimposed edge map is a factor here. Further analysis on this is left for future work.

Figure 8 also helped to answer our questions about the effects of downsampling in general on the performance of our feature vectors. Surprisingly, little performance was lost with our chosen "default" downsampling factor of 6. Beyond this, however, we see accuracy begins to fall off as more and more information is lost. It is interesting to note, however, that even with a downsampling factor of 24 (that is, a 24x24 = 576 factor reduction in the number of pixels, down to a tiny 9x7 image), we can still do roughly 50x better than random guessing on the "Night Before Christmas" dataset (random guessing gives us an accuracy of 1/114, while using the Average type feature vector we achieve 50/114). In other words, there's still a lot of information about the original 21,375 pixels packed into the downsampled 27. This observation leads us to believe that other feature set reduction methods, such as PCA, may yield even better results than those we achieved having chosen downsampling rather arbitrarily.

Even so, with simple downsampling, our roughly 80% accruacy on a 114 word dataset with a downsampling factor of 6 and only 10 samples per word was very surprising. On our datasets, however, we noted something a little bit simplistic about them, which was that despite having so many unique words, these words were always typed in the same order. For instance in the "Night Before Christmas" dataset, the poem was typed 10 times, and always from start to finish. Hence a word like "night" is always preceeded by "the" and succeeded by "christmas". Therefore if there were to be any effect of neighboring words on the typing of the word "night", our dataset would not be capturing that at all. To explore the question of "neighbor effects" on word typing, we typed

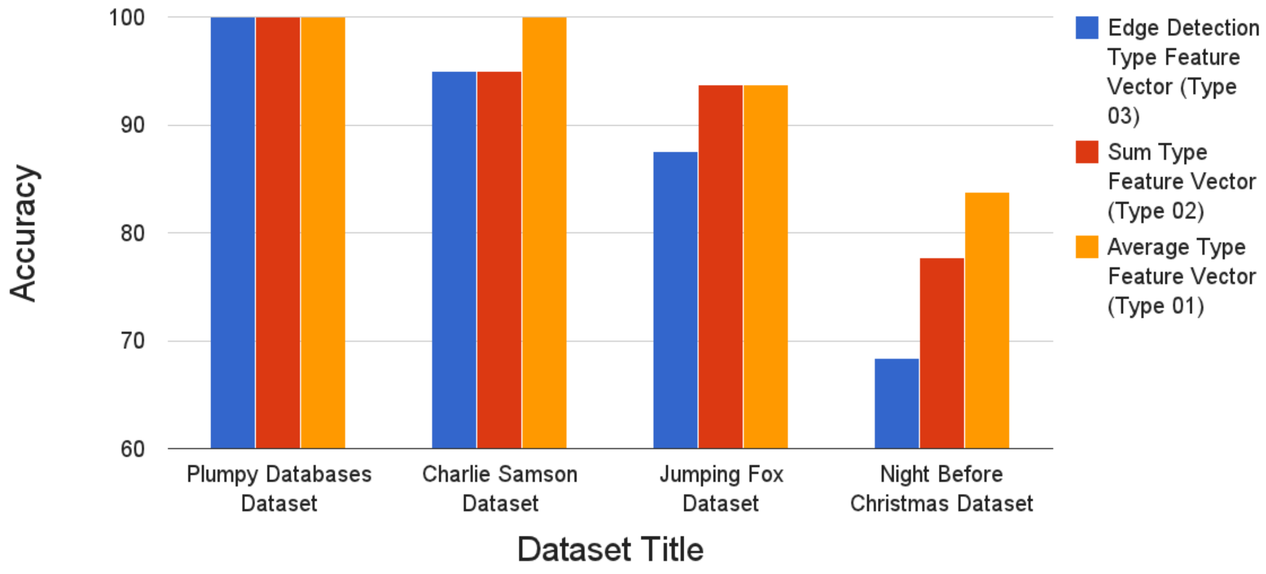


Fig. 7: Feature vector performance comparison across datasets

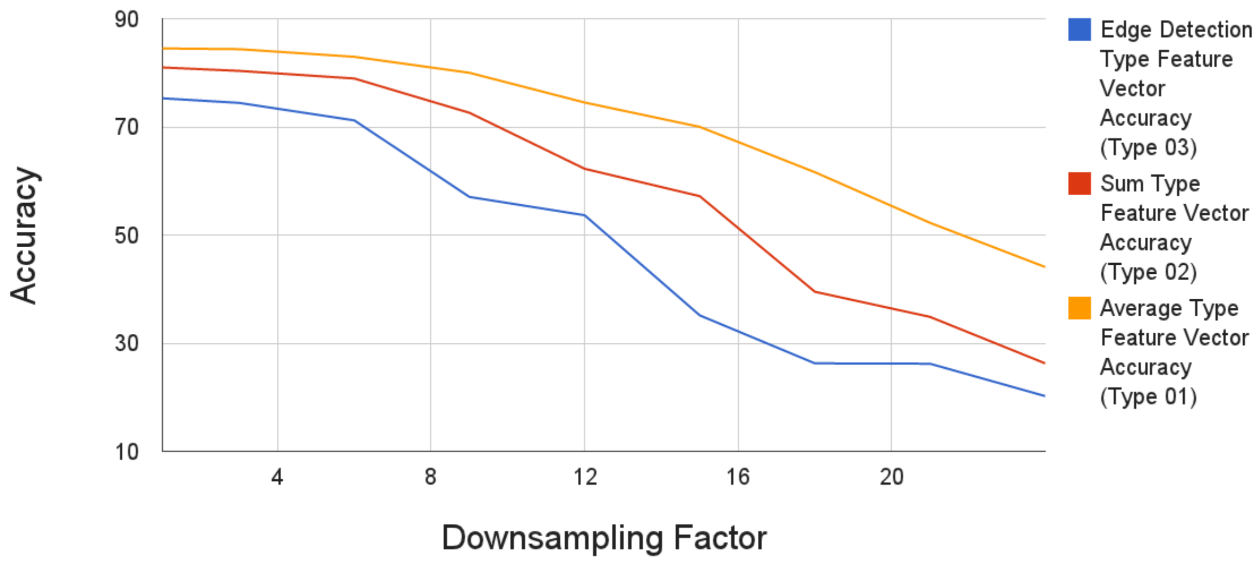


Fig. 8: Effects of downsampling on the ”Night Before Christmas” dataset

the “Night Before Christmas” poem backwards (“... christmas before night the twas”) and used it as a testing set against the original forward typed dataset, treated as a training set. The results are shown in Figure 9 and reveal clearly that indeed the hands type words differently depending on what word comes before and what word comes after. Still, though, in the best case we measured with the Averaging type feature vector, we were able to achieve roughly 65% accuracy, giving us hope that there is some significant amount of information still contained in the feature vectors that is independent of neighboring words. Exploring this, too, is left for future study.

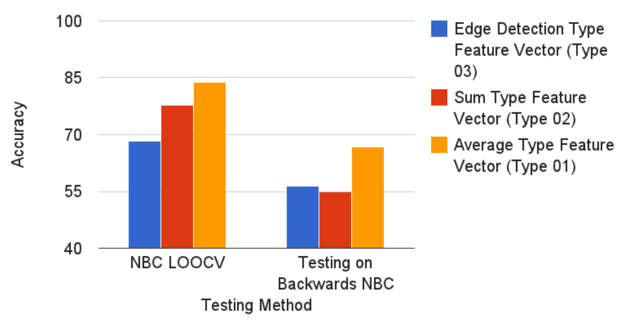


Fig. 9: The "Neighboring Words" Effect

VII. Future Work

There are many avenues to explore for future work with the Learning Keyboard. First, the "neighboring words" effect needs to be investigated with the hope of developing feature vectors that are able to extract out the neighbor independent aspects of word typing. One idea with respect to this might be to throw away a few video frames at the beginning and end of the typing of a word, leaving only the motion of the hands that occurred in the middle of the word. This may work well for long words, but short words like 'a' and 'I' may require other techniques.

Secondly, the present implementation of the system does not support the "delete" key, in the sense that if a word is mistyped, partially deleted, and then finished correctly, the system treats that as an unknown word and throws it out of the collected data. To make the Learning Keyboard a useable system for real data entry, a method for handling corrected words would be required.

Lastly, and also towards making the system practically usable, a method for parsing out words from the depth map stream without any a-priorio knowledge of keypresses would be required. Presently the system parses out words from the datastream, and hands these segments one word at a time to the SVM prediction system to guess which word was typed... but in reality such a Learning Keyboard would need to first guess where words are being typed in the video stream (perhaps by detecting presses to the spacebar), and then guessing which word was typed based on the spliced video.

Beyond these, there are many exciting avenues of future exploration including tuning various aspects of the system, using PCA instead of downsampling, making the system independent of camera placement, and even seeing if it would be possible for a machine to learn what a user is typing without ever seeing keylog data (that is, use unsupervised learning algorithms along with marchov models of typed words to make accurate estimates of what the user is typing). For spy applications it may not be critical to know precisely word for word what the user typed, but rather to understand the general meaning of their typed message.

VIII. Conclusion

With an Xbox Kinect, an open source image processing library, and an off-the-shelf support vector machine we were able to successfully make a first attempt at building the Learning Keyboard and show some interesting first results, including over 80% accuracy on a 114 unique word dataset with 10 samples per word. Though the simplification was made that words were typed in the same order each time, we were able to show with some initial findings that our chosen feature vectors still maintained an information component that was indepdent of neighboring words, and gives us hope that there may be other feature vector generating schemes that better minimize the "Neighboring Words" effect. We believe that with larger datasets, better tuned feature vectors, and perhaps an attempt at tuning the default SVM parameters for LIBSVM, we should be able to achieve even better results than these. Perhaps one day the Learning Keyboard or machine learning techniques like the ones used here will indeed be able to make the presence of a physical keyboard a mere memory of the past, but a story told to grandchildren.

"Twas the night before Christmas and all through the house, not a keyboard was clacking, not even a mouse".

Acknowledgments

We would like to thank Chris Lengerich for providing useful advice for this project.

References

[1] OpenKinect driver for Mac, http://openkinect.org/wiki/Main_Page

[2] TKinter library http://wiki.python.org/moin/TkInter

[3] Canny http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/canny_detector/canny_de

[4] Chang, Chih-Chung and Lin, Chih-Jen, *LIBSVM: A library for support vector machines* ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27,2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.