Protractor: A Fast and Accurate Gesture Recognizer

Yang Li
Google Research
1600 Amphitheatre Parkway
Mountain View, CA 94043
yangli@acm.org

ABSTRACT
Protractor is a novel gesture recognizer that can be easily implemented and quickly customized for different users. Protractor uses a nearest neighbor approach, which recognizes an unknown gesture based on its similarity to each of the known gestures, e.g., training samples or examples given by a user. In particular, it employs a novel method to measure the similarity between gestures, by calculating a minimum angular distance between them with a closed-form solution. As a result, Protractor is more accurate, naturally covers more gesture variation, runs significantly faster and uses much less memory than its peers. This makes Protractor suitable for mobile computing, which is limited in processing power and memory. An evaluation on both a previously published gesture data set and a newly collected gesture data set indicates that Protractor outperforms its peers in many aspects.

Author Keywords
Gesture-based interaction, gesture recognition, template-based approach, nearest neighbor approach.

ACM Classification Keywords
H5.2. [Information interfaces and presentation]: User interfaces. I5.2. [Pattern recognition]: Design methodology – Classifier design and evaluation.

General Terms
Algorithms, performance.

INTRODUCTION

An important topic in gesture-based interaction is recognizing gestures, i.e., 2D trajectories drawn by users with their finger on a touch screen or with a pen, so that a computer system can act based on recognition results. Although many sophisticated gesture recognition algorithms (e.g., [2]) have been developed, simple, template-based recognizers [4, 5] often show advantages in personalized, gesture-based interaction, e.g., end users defining their own gesture shortcuts for invoking commands. First I offer my insight into why template-based recognizers may be superior for this particular interaction context. I then focus on Protractor, a novel template-based gesture recognizer.

Template-based recognizers essentially use a nearest neighbor approach [3], in which training samples are stored as templates, and at runtime, an unknown gesture is compared against these templates. The gesture category (or the label) with the most similar template is used as the result of recognition, and the similarity implies how confident the prediction is. These template-based recognizers perform limited featurization, and a stored template often preserves the shape and sequence of a training gesture sample to a large degree. These recognizers are also purely data-driven, and they do not assume a distribution model that the target gestures have to fit. As a result, they can be easily customized for different domains or users, as long as training samples for the domain or user are provided.

In contrast, recognizers that employ a parametric approach [3] often operate on a highly featurized representation of gestures and assume a parametric model that the target gestures have to fit. For example, the Rubine recognizer [2] extracts a set of geometric features from a gesture such as the size of its bounding box. It uses a linear discriminate approach to classify gestures that assumes the featurized gestures to be linearly separable. These parametric recognizers can perform excellently when the target gestures truly fit the assumed model. However, if not, these recognizers may perform poorly.

For personalized, gesture-based interaction, it is hard to foresee what gestures an end user would specify and what the distribution of these gestures will look like. In addition, since an end user is often willing to provide only a small number of training samples, e.g., one sample per gesture category, it is hard to train a parametric recognizer that often has a high degree of freedom with such sparse training data. In contrast, template-based recognizers are well suited for this situation.

However, since a template-based recognizer needs to compare an unknown gesture with all of stored templates to make a prediction, it can be both time and space consuming, especially for mobile devices that have limited processing power and memory. In the remainder of this paper, I introduce Protractor[1], a novel recognizer for gesture recognition that outperforms its peers in many accounts including recognition speed, accuracy and gesture variation.

PROTRACTOR

Protractor employs a nearest neighbor approach. For each gesture (either an unknown gesture or a training sample), Protractor preprocesses it into an equal-length vector. Given an unknown gesture, Protractor searches for similar gesture templates by calculating an optimal angular distance between the unknown gesture and each of the stored templates. Protractor uses a novel closed-form solution to calculate such a distance, which results in significant improvements in accuracy and speed. Protractor also recognizes gestures that are both invariant and sensitive to orientation, as well as gestures with different aspect ratios.

Preprocessing

Protractor's preprocessing is similar to the \$1 recognizer's [4], but with several key differences in handling orientation sensitivity and scaling. This process is intended to remove irrelevant factors, such as different drawing speeds, different gesture locations on the screen, and noise in gesture orientation. The preprocessing transforms the 2D trajectory of a gesture into a uniform vector representation.

To do so, Protractor first resamples a gesture into a fixed number, N, equidistantly-spaced points, using the procedure described previously [4], and translate them so that the centroid of these points becomes (0, 0). This step removes the variations in drawing speeds and locations on the screen.

Next, Protractor reduces noise in gesture orientation. The orientation information of a gesture can be useful or irrelevant, depending on the application. Protractor gives the developer an option to specify whether it should work in an orientation-invariant or -sensitive way. When Protractor is specified to be orientation invariant, it rotates a resampled gesture around its centroid by its indicative angle, which is defined as the direction from the centroid to the first point of the resampled gesture [4]. This way, all of the templates have zero indicative orientation.

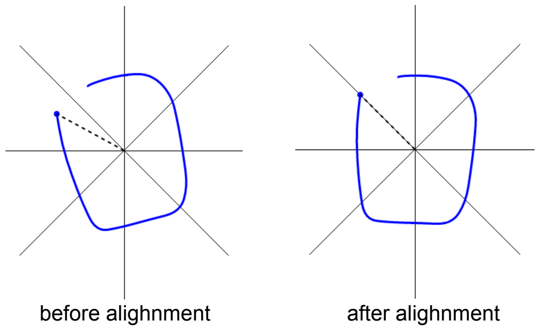

Figure 1. When Protractor is specified to be orientation sensitive, it aligns the indicative orientation of a gesture with the closest direction of the eight major orientations.

[1] Pseudocode is available at http://yanglisite.net/protractor.

However, when Protractor is specified to be orientation sensitive, it employs a different procedure to remove orientation noise. Protractor aligns the indicative orientation of a gesture with the one of eight base orientations that requires the least rotation (see Figure 1). The eight orientations are considered the major gesture orientations [1]. Consequently, Protractor can discern a maximum of eight gesture orientations. Since Protractor is data-driven, it can become orientation-invariant even if it is specified to be orientation-sensitive, e.g., if a user provides gesture samples for each direction for the same category,

Based on the above process, we acquire an equal-length vector in the form of (x_1, y_1, x_2, y_2, …, x_N, y_N) for each gesture. For each gesture, the preprocessing only needs to be done once. In the current design, Protractor uses $N = 16$, which allows enough resolution for later classification. 16 points amount to a 32-element vector for each gesture, which is ¼ of the space required by previous work for storing a template [4]. Note that Protractor does not rescale resampled points to fit a square as the \$1 recognizer does [4], which preserves the aspect ratio of a gesture and also makes it possible to recognize narrow (or 1-dimensional) gestures such as horizontal or vertical lines. Rescaling these narrow gestures to a square will seriously distort them and amplify the noise in trajectories.

Classification by Calculating Optimal Angular Distances

Based on the vector representation of gestures acquired by the above process, Protractor then searches for templates that are similar to the unknown gesture. To do so, for each pairwise comparison between a gesture template t and the unknown gesture g, Protractor uses the inverse cosine distance between their vectors, v_t and v_g, as the similarity score S of t to g.

$$S(t,g) = \frac{1}{\arccos \frac{v_t \cdot v_g}{|v_t||v_g|}} \tag{1}$$

The cosine distance essentially finds the angle between two vectors in an n-dimensional space. As a result, the gesture size, reflected in the magnitude of the vector, becomes irrelevant to the distance. So Protractor is inherently scale invariant. The cosine distance of two vectors is represented by the dot product of the two vectors (see Equation 2) divided by the multiplication of their magnitudes (see Equation 3).

$$v_t \cdot v_g = \sum_{i=1}^{n} \left(x_{ti} x_{gi} + y_{ti} y_{gi} \right) \tag{2}$$

$$|v_t||v_g| = \sqrt{\sum_{i=1}^{n} \left(x_{ti}^2 + y_{ti}^2 \right)} \sqrt{\sum_{i=1}^{n} \left(x_{gi}^2 + y_{gi}^2 \right)} \tag{3}$$

However, it can be suboptimal to evaluate the similarity of two gestures by just looking at the angular distance calculated by Equation 1. As discussed in the previous section, Protractor acquires the vector representation of a gesture by aligning the gesture's indicative orientation. Since the indicative angle is only an approximate measure of a gesture's orientation, the alignment in the preprocessing cannot completely remove the noise in gesture orientation. This can lead to an imprecise measure of similarity and hence an incorrect prediction. To address this issue, at runtime, Protractor rotates a template by an extra amount so that it results in a minimum angular distance with the unknown gesture and better reflects their similarity. Previous work [4] performs similar rotation to find a minimum mean Euclidean distance between trajectories. However, it used an iterative approach to search for such a rotation, which is time-consuming and the rotation found can be suboptimal.

In contrast, Protractor employs a closed-form solution to find a rotation that leads to the minimum angular distance. As we will see in the experiment section, this closed-form solution enables Protractor to outperform previous recognizers in both recognition accuracy and speed. Here I give the closed-form solution.

Since we intend to rotate a preprocessed template gesture t by a hypothetical amount θ so that the resulting angular distance is the minimum (i.e., the similarity reaches its maximum), we formalize this intuition as:

$$\theta_{optimal} = \underset{-\pi \le \theta \le \pi}{\arg\min} \left(\arccos \frac{v_t(\theta) \cdot v_g}{|v_t(\theta)||v_g|} \right) \qquad (4)$$

$v_t(\theta)$ represents the vector acquired after rotating template t by θ. Note that this is on top of the alignment rotation that is performed in the preprocessing. As we intend to minimize the cosine distance with respect to θ, we find

$$\frac{d\left(\arccos \frac{v_t(\theta) \cdot v_g}{|v_t(\theta)||v_g|} \right)}{d\theta} = 0 \qquad (5)$$

Solving Equation (5) gives the following solution:

$$\theta_{optimal} = \arctan \frac{b}{a} \qquad (6)$$

where a is the dot product of v_t and v_g (see Equation 2) and b is given in Equation 7.

$$b = \sum_{i=1}^{n} \left(x_{ti} y_{gi} - y_{ti} x_{gi} \right) \qquad (7)$$

With $\theta_{optimal}$ calculated, we can easily acquire the maximum similarity (the inverse minimum cosine distance) between the two vectors. We then use this similarity as the score for how well gesture template t predicts the unknown gesture g. The gesture template that has the highest score becomes the top choice in the N-best candidate list.

PERFORMANCE EVALUATIONS

To understand how well Protractor performs, I compared it with its closest peer, the $1 recognizer [4], by repeating the same experiment on the same data set where the $1 recognizer showed advantages over both the Rubine [2] and the DTW recognizers [5]. The data set includes 4800 samples for 16 gesture symbols collected from 10 participants (e.g., a star) [4]. The experiment was conducted on a Dell Precision T3400 with a 2.4GHz Intel Quad Core^{TM}2 CPU and 4 GB memory running Ubuntu Linux.

Overall, Protractor and the $1 recognizer generated a similar error rate curve in response to different training sample sizes (see Figure 2). Although the overall Poisson regression model for predicting errors was statistically significant (p<.0001), the major contributor to this significance is the training sample size and there was no significant difference between the recognizers (p=.602).

However, Protractor is significantly faster than the $1 recognizer (see Figure 3). Although the time needed for recognizing a gesture increases linearly for both recognizers as the number of training samples grows, the $1 recognizer increases at a much rapid rate. For example, when 9 training samples are used for each of the 16 symbols, the $1 recognizer took over 3 ms to recognize a gesture, while it took Protractor less than ½ ms to do so.

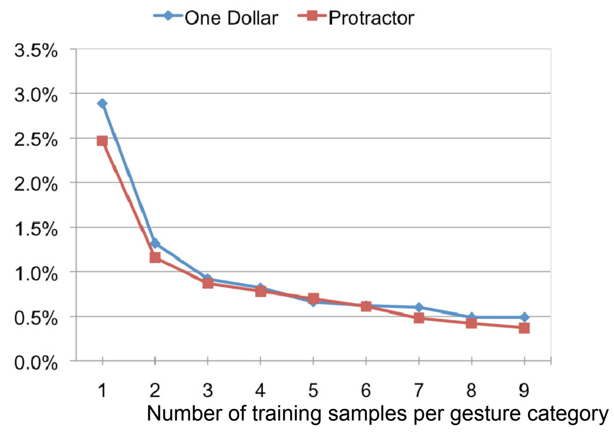

Figure 2. The error rates of both the $1 recognizer and Protractor decrease as more training samples are used.

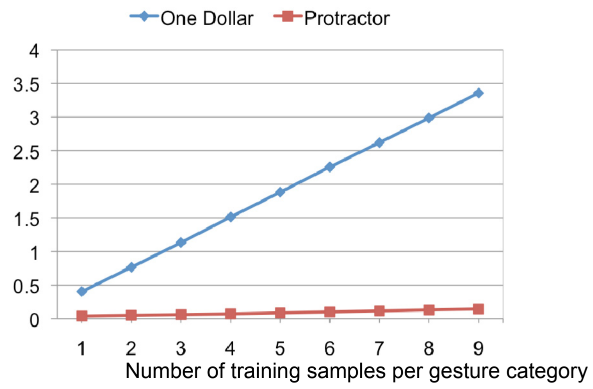

Figure 3. The milliseconds needed for recognizing a gesture grows as the number of training samples increases. Protractor runs significantly faster than the $1 recognizer.

To better understand the impact of the time performance of these recognizers on mobile devices, I repeated the above experiment on a T-Mobile G1 phone running Android. When 9 training samples were used for each of the 16 gesture symbols, it took the $1 recognizer 1405 ms (std = 60 ms) to recognize a gesture, while it only took Protractor 24 ms (std = 7 ms) to do so. The time cost of the $1 recognizer grew rapidly as the number of training samples increased (mean = 155 ms/16 samples, std = 2ms). As part of a process of continuous learning, a template-based recognizer needs to constantly add new training samples generated by user corrections. However, the rapidly growing latency of the $1 recognizer makes it intractable to do so. In contrast, the time cost of Protractor grew at a much slower pace (mean = 2 ms/16 samples, std = 1 ms).

To understand how both recognizers perform on a different data set, I tested them on a larger gesture set that includes 10,888 single-stroke gesture samples for 26 Latin alphabet letters. They were collected from 100 users on their own touch screen mobile phones. Similar to the previous experiments, I randomly split the data of each user for training and testing based on different training sizes. Since each alphabet had at most 5 samples from each user, we could only test training sizes from 1 to 4. Overall, both recognizers performed less accurate on this data set than they did on the previous 16-symbol data set (see Figure 4). The loss in accuracy was primarily because the new data set is more complex as it includes 26 gesture categories, compared to 16 symbols of the previous data set. This gesture data was also collected in a more realistic situation than the laboratory environment that was used previously [4]. However, we see more rapid improvement of both recognizers as the training size increases (see Figure 4). In particular, Protractor performed significantly more accurate than the $1 recognizer on this data set ($p < .0001$).

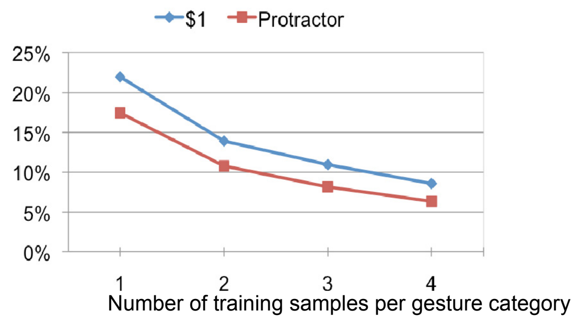

Figure 4. The error rates of both recognizers on an alphabet gesture set collected from 100 mobile phone users.

DISCUSSIONS

As Protractor can recognize variation in gesture orientation and aspect ratio, there is also a risk that it might pick up noise in these variations. However, based on the above experiments, Protractor is as accurate as the $1 recognizer on the smaller data set (4800 samples / 16 categories / 10 users) and is significantly more accurate on the larger data set (10,888 samples / 26 categories / 100 users).

In addition to specifying whether Protractor should be orientation sensitive, a developer can also specify how sensitive it should be to orientation, e.g., whether two or four directions are allowed, which will bound the solution of Equation 6. At eight directions, Protractor started to pick up some noise in orientation, which led to a significant increase in error rates (see Figure 5).

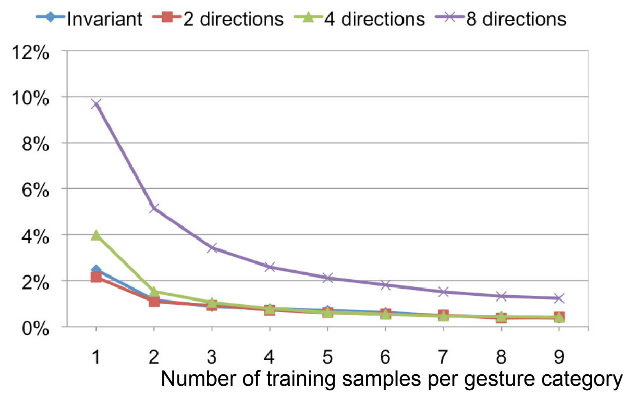

Figure 5. The error rates of Protractor for different orientation sensitivities based on the tests with the 16-symbol data set.

As a nearest neighbor recognizer needs to load all of the training samples into memory before it can make a prediction, the amount of space needed is a critical factor, especially on mobile devices. Protractor uses ¼ of the space that is required by the $1 recognizer. With the closed-form solution, Protractor can also search through stored templates over 70 times faster than $1 on a T-Mobile G1.

CONCLUSION

I designed Protractor, a template-based, single-stroke gesture recognizer that employs a novel closed-form solution for calculating the similarity between gestures. I evaluated Protractor on different data sets and platforms and found that it outperformed its peer in many aspects, including recognition accuracy, time and space cost, and gesture variation. In addition, I also discussed my insight into why template-based recognizers in general have gained popularity in personalized, gesture-based interaction, other than their obvious simplicity.

REFERENCES

1. Kurtenbach, G. and Buxton, W. The limits of expert performance using hierarchical marking menus. *CHI'93*. 1993. p. 35-42.
2. Rubine, D., Specifying gestures by example. *ACM SIGGRAPH Computer Graphics*, 1991. **25**(4): p. 329-337.
3. Russell, S. and Norvig, P., Artificial Intelligence: A Modern Approach. 2 ed. 2002. *Prentice Hall*.
4. Wobbrock, J.O., Wilson, A. and Li, Y., Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes, *UIST'07*. 2007. p. 159-168.
5. Zhai, S. and Kristensson, P.-O. Shorthand writing on stylus keyboard. *CHI'03*. 2003. p. 97-104.