Setalyzer

Adam Lerner & Will Scott

Introduction & Motivation

Setalyzer is a program for solving a common problem when playing the Set card game: no player sees a legal move, but nobody is confident that a valid move does not exist. To address this problem, we built an Android application, which uses a set of computer vision techniques to identify and then augment the game with remaining moves.

The Set game itself is a computationally tractable problem space. There will typically be 12 cards in play, and a move consists of matching three cards. This means that there are 12 choose 3 possible moves at any given time, which can be fully explored very quickly. The harder problem is correctly identifying the cards in play quickly and without user interaction, so that the application can be useful in context.

Approach

We built Setalyzer as an Android application. This means that Setalyzer is programmed in pure Java, and feature extraction and classification of camera data must be calculated within tight memory and time bounds. We rely on the underlying image representation used by Android, since transformations of the image are expensive, which surprisingly meant there was very sparse existing machinery for us to build upon, and resulted in much of our time being spent building the system for extracting, manipulating and classifying images.

Our analysis process is split into two main components, with a training phase split into two parts and a live evaluation phase. The resulting application performs the following steps in this live evaluation: we take an image from the phone's camera, and first segment and then classify that image. Segmentation identifies cards in the image, and reconstructs them as face-on rectangles. Classification identifies features in a card image, and then evaluates them using a pre-trained classifier. After classification, results are drawn back on the device screen.

In order to train the classifier, we created a desktop training program to allow us to hand label a test corpus of images we took. Once these images are labeled, we can extract features from them to produce the attribute relation files used by the Weka classification system. Since the feature extraction code is platform specific (it relies on image manipulation of the native Android Bitmap image format, which is a java interface to ARM native code), determining features on the test image set requires extraction on the phone after labeling, to ensure that we have representative features for those images.

Our classifier must be robust on four axes. Each card in the game has one value for each of four trinary characteristics - color, fill pattern, count, and shape.

Classification and Features

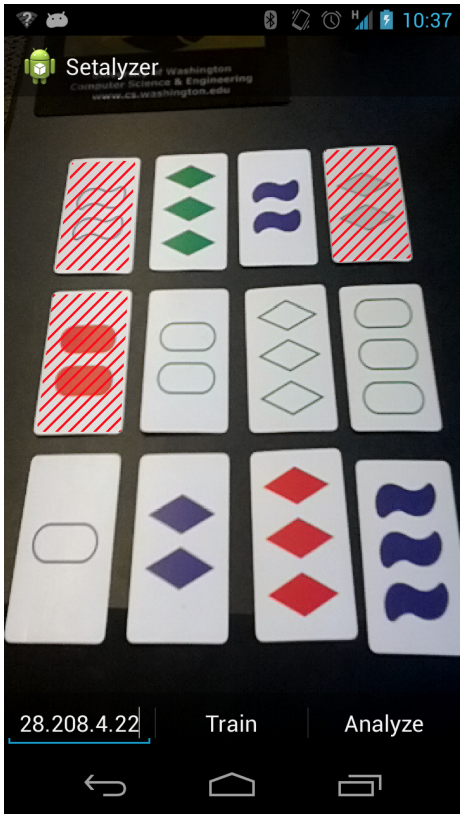
Our classification algorithm takes as input a series of features extracted from a tightly cropped image of a card and uses those features as inputs to a trained classifier. Using our current features, we ran preliminary tests using a variety of classifier types provided by Weka, including Naive Bayes, Bayes Nets, IB1 (a nearest-neighbor classifier), several types of meta boosting classifiers, and decision trees. None of these approaches produced satisfactory results even over a small set of test-cases, with the issue coming from a low signal in our feature extraction, rather than the classification technique.

For features, we currently extract 12 features from each card image. Nine of those features are pixel color value averages. Those features are derived by sampling each feature from a different portion of the image, including the whole image, diagonal transects, and the center of the image. The final 3 features are built from a distance metric based upon SURF (Speeded Up Robust Features). We calculate SURF features for the image to be classified, which provides a list of points of interest in the image. A strength of SURF for this application is that it encourages these points to be calculated similarly for images which vary in both rotation and size, allowing our features to be scale and rotation invariant.

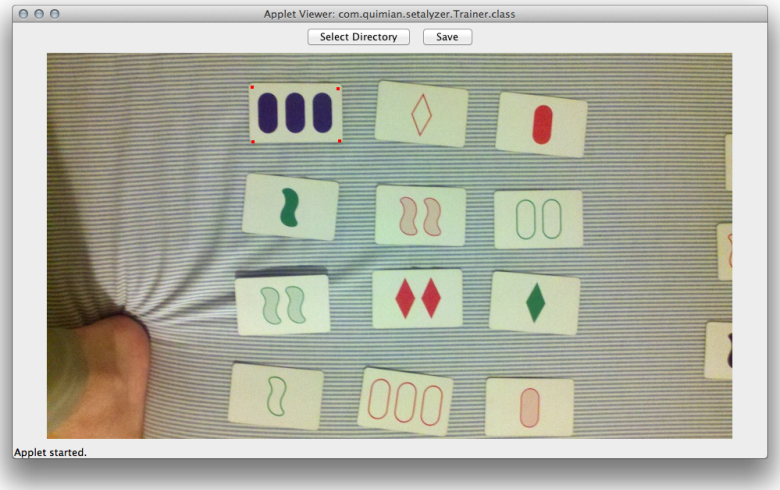
We also calculate SURF features for 3 canonical examples of cards. Then we apply a greedy association algorithm which matches similar SURF points between our example and those canonical cards. From this we derive 1 feature per canonical card: the number of matches which the association algorithm finds between the features of the example and the canonical card. The example card's similarity to known cards seems that it should be a likely candidate for a useful feature.

Between these features, we expected that the average color value features would help characterize the color property of the cards, as well as possibly the fill pattern and count, since for example central pixel values will be on average lighter in a card which has a count of 2 (and thus no symbol directly centered in the card). We expected that the distance metric feature should primarily capture count and shape, as it is invariant with color and SURF features don't seem to vary significantly with fill pattern (they appear largely on the edges of the symbols). We discuss a number of possible improvements to our features at the end of the evaluation section, below.

Evaluation



The client Interface. Setalyzer marks cards which form a set. The bottom toolbar allows the phone to act as a feature extractor for previously labeled images.

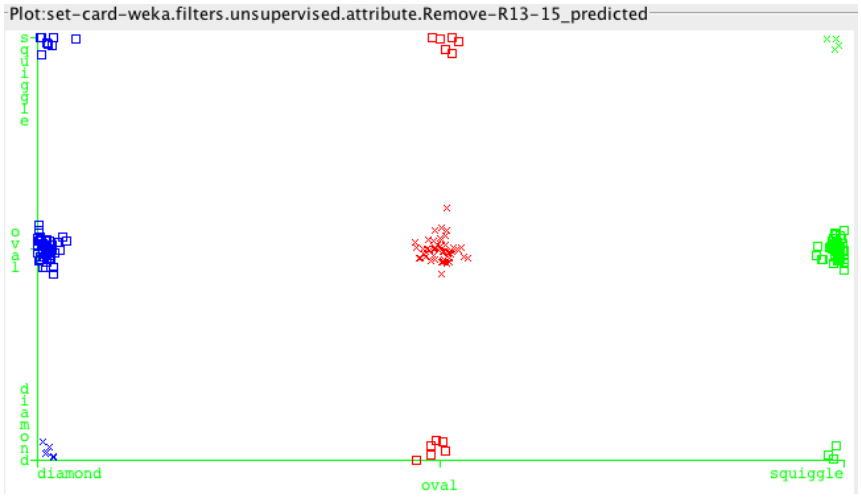


The Setalyzer training Interface, allowing labeling of captured images. The labels are stored, and a second application computes features for each labeled card.

An important piece of the Setalyzer system is allowing it to run on an Android handset. To accomplish this, we looked at the performance for both training and execution of samples. We will add that the code uses several idioms which are known to be expensive in android, based on how the existing Boof library extracts features - for example image buffers are recreated several times, which is highly discouraged, because repeated interactions with Android heap management is much slower than reusing a single buffer. Thus we consider these numbers primarily as diagnostic for where to focus optimization efforts rather than as characterizing the runtime our application might eventually achieve.

Feature generation for the 179 test images we used took approximately 30 minutes (approximately 10 seconds per image). Part of this time was spent on data transfer overhead, since the full image was transferred to the phone for each calculation.

Full processing of an image currently takes 40 seconds when cards are present, which is unacceptably long. ¾ of the time is currently spent on SURF feature extraction, which is by far the most expensive part of the analysis. Segmentation is completed in well under a second, and much of the rest of the time is spent in the overhead of transforming detect cards, and converting image formats.



Visual representation of the confusion matrix for identifying shapes using a Naive Bayes Classifier against 179 training samples. The classifier preferred Ovals, although the samples were evenly distributed in shape, correctly classifying 33.5% of test instances in Cross-Validation.

The accuracy of each of the classifiers we tried ranged between 30% and 35% for each trinary characteristic. For example, the Bayes Net classifier had 33.2% accuracy, and a Bagging meta classifier was able to reach 36.3% accuracy for the shape property. This is not significantly better than a random classifier would have performed, and we believe this indicates that our features do not provide sufficient signal to identify cards.

We have considered a number of additional features and improvements to current features. We currently use only 3 canonical cards which do not fully represent the space of possible cards. An extension would be to calculate the distance metric from each of the 81 possible cards in the deck, or from a well distributed subset of the deck if calculating all 81 distance features takes too long given the computational resources of an average Android device. We might also tweak the distance metric by incorporating the quality of point matching and the location of the points matched. Other possible features could include those derived from corner or edge detection algorithms, color distribution from the overall image or of all cards extracted from the containing image (to account for lighting variation), and other simpler features such as total proportion of "white" vs "dark" pixels in the image or the number of contiguous regions of each type of pixels (since segmentation involves computing a binary threshold of the image, we can reuse the results of those computations as classification features for free).

Conclusions & Future Works

One of the results of this project which we are most excited about is that we were able to build a native Android system for evaluating feature extraction routines. OpenCV is a library providing existing feature extractors, but it does not support Android natively, nor does it provide an integrated mechanism for evaluating the significance of extracted features. As part of Setalyzer we ended up building a generic component allowing for iterative evaluation of feature extractors, and while we were unsuccessful in finding appropriate features for our vision problem, we think the evaluation framework can become a valuable artifact.

The future work remaining for Setalyzer rests on determining a set of tractable, high quality features. Additionally, there is a restructuring possible in the application where instead of evaluating the features in each camera frame individually, we loosen the time requirement for evaluation and instead attempt to do a simple motion tracking algorithm between frames to update the overlay. That change will allow for a less jerky user experience, but requires an additional layer of computation.

Appendix: Logistics

This project was naturally parallelizable. Will worked primarily on the training system and the pipeline for evaluating feature detection, while Adam worked on the primary execution pathway of segmentation and feature extraction.

Setalyzer uses Weka for classification, and the BoofCV library for some feature extraction algorithms. BoofCV was chosen because it was a native Java codebase which runs on Android, unlike the more powerful and commonly used OpenCV. The downside is that it is much less powerful, and provides only a minimal set of tools like edge extraction and thresholding.

To run setalyzer, checkout the source from https://github.com/willscott/setalyzer. The setalyzer project contains a functioning Android Application, which can be installed on Android devices using the eclipse android plugin. Alternatively, packed apk binaries are uploaded to github.

Appendix: References

Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features." Computer Vision–ECCV 2006 (2006): 404-417.

BoofCV: http://boofcv.org/index.php?title=Main_Page

Canny, John. "A computational approach to edge detection." Pattern Analysis and Machine Intelligence, IEEE Transactions on 6 (1986): 679-698.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten (2009); The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.

W.C. Chen, Y. Xiong, J. Gao, N. Gelfand, and R. Grzeszczuk, "Efficient Extraction of Robust Image Features on Mobile Devices." ISMAR '07