vision challenge

team: later than planned
eric conner
zach galant
jeremy keeshin

Data Structures Overview

For our final project, we mainly used data structures from the OpenCV library. We use an array of CvBoost objects as our boosted model to classify each of the objects on the screen. We also use a CObjectList called previous objects to store all of the objects found on the previous frame to track using optical flow. In addition we use save the previous frame as in IplImage* so we can compute flow between the two frames.

In addition to these data structures we wrote two extra binaries to help extract features from the training examples:

./makeDict -d dictFileName.xml -cf true /afs/ir/class/cs221/vision/data/vision_all

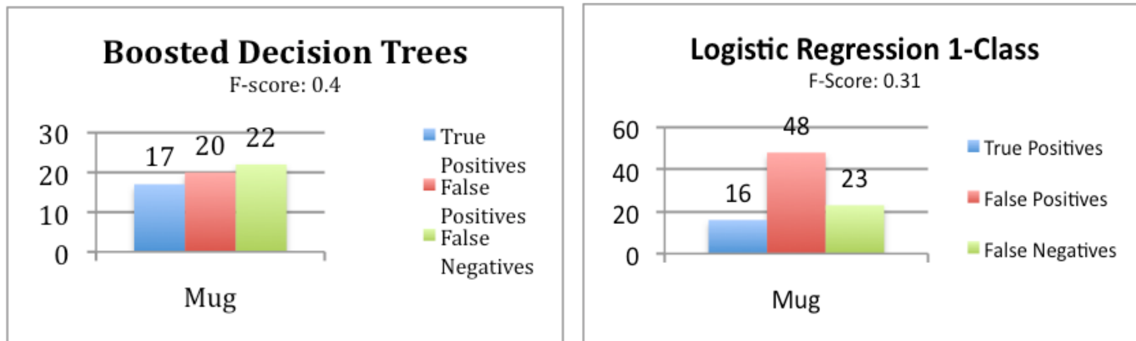
usage: -d <filename> :: filename to output the feature dictionary

-cf true :: use corner features, a flag that sets whether to use corner detection to locate feature fragments or random regions

./pruneFeatures -i input.xml -o output.xml /afs/ir/class/cs221/vision/data/vision_all

Boosted Decision Trees

We first implemented boosted decision trees because we knew the algorithm would outperform logistic regression, which can only linearly classify the features. We used the OpenCV Boost class to train decision trees based on the mug fragment features given for the milestone:



From the graphs, the initial improvement over logistic regression was modest, but noticeable. We had not tweaked the decision trees much but had tweaked logistic regression extensively for the milestone and still decision trees outperformed regression. Our first experiments included decision stumps with around 200 rounds of boosting, but the classifier did make consistent decisions about how to label the mugs. This inconsistency suggested adding more features to each tree so we tried trees of depth two with 200 boosting rounds. After running experiments with 200 rounds at depth two, our false negative count rose consistently, suggesting some over fitting. Finally we found that 100 rounds of boosting with trees of depth two gave the best performance.

Throughout these experiments on boosting parameters we tried training on different sets of the training data. In general we found that training on all of the "other" data resulted in over fitting while training on a fifth or less under fit. Generally our best results came from training on all of the positive examples and one third of the negative training examples.

Conclusions from one class boosted trees:

- Train on 1/3 of negative examples for each positive class
- Use 100 rounds of boosting with trees of depth two

Hough Transform For Circles

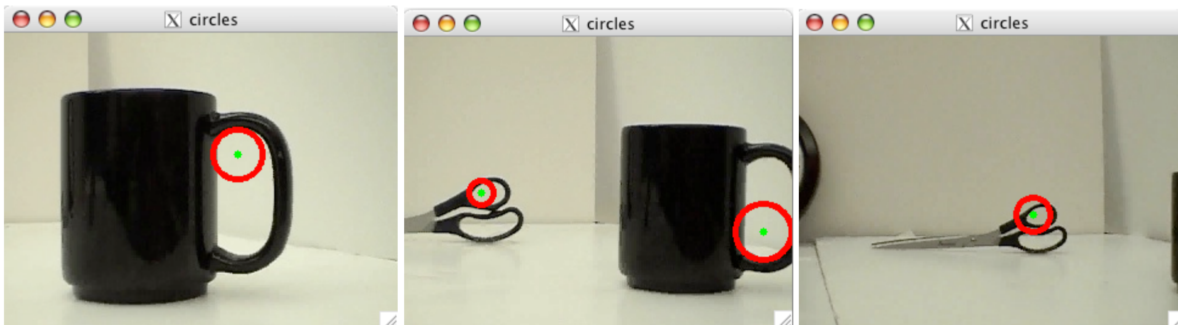
One of our extensions was to use the Hough Transform for Circles to help find clocks. The main intuition behind this is that clocks are circles (or at least this is the case for the clocks we are looking for; I don't believe we needed to find watches or iHomes), and if the Hough Transform finds circles, then it should in theory find clocks that are circles. This surprisingly simple intuition led to some of the most successful results in our coding and testing process.

People may think this idea is too simple or silly to work, but it had some of the most positive test results of our entire process. For the other objects we used our boosted decision tree classifier, but since there was such a clear mapping from Hough Circles to clocks, this proved to be the best way.

Here is the process we went through:

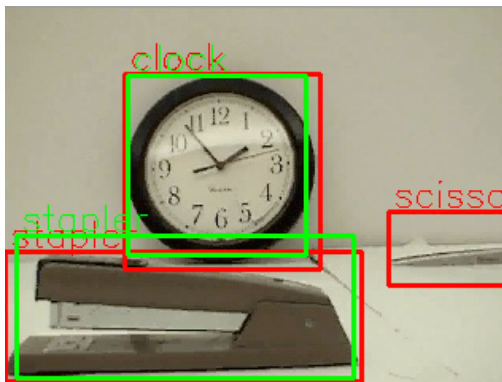
First we used the openCV Hough Circles algorithm to find the circles on the frame. In the attached screenshot you can see that it finds the clocks almost exactly. When we looked at this image, we realized that sometimes it finds the inner circle on the clock, so we added a small buffer to account for this.

Next, we would draw a rectangle circumscribed about the Hough circle and call that our clock. As perfect as this seems, there were several other obstacles in our path. In the screenshot below you can see that the Hough Circles algorithm finds all things that it thinks are circles, which means that it also finds mug handles and scissors handles. If we simply labeled everything that was a circle a clock, we would get lots of false positives.



Circles don't always lead to clocks. We had to account for other circles detected on mugs and scissors.

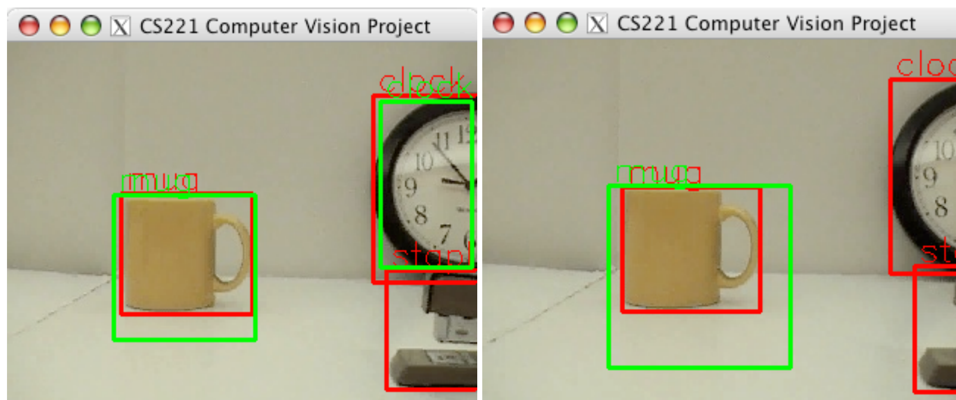
We needed to find a way to accommodate our data for the Hough Circles in with our Boosted Decision tree classifier. When we tested our boosted decision tree classifier, it would find clocks, but it would not be extremely accurate. Actually, it would find several clocks. However, if the Hough Circles algorithm found a clock, it would be exactly on. We had the idea that if we found a clock with our classifier, we would check if the Hough Circles algorithm could find a circle that overlaps with the clock by some threshold, and if it does, then replace the classifier's prediction with the Hough Circle algorithm prediction.



This was an extremely successful approach, so when we find clocks, the Hough Circle algorithm will draw them exactly. You can see in this screenshot the Hough Circles accommodation we used in action.

With this, we would achieve up to 0.83 F-Score on clocks. The evaluation said that we still had false positives and false negatives, but if you watched the videos you would not think that. We examined the xml file and the differences are very hard to find. However, we thought 0.83 was a good score on clocks.

Another obstacle we faced with this method was that we wanted our Hough Circles algorithm to basically override all of the other post-processing steps and tracking steps that we were doing because it was so clearly finding then. We then essentially turned off optical flow for clocks as well as our rectangle-averaging algorithm.



If you get rid of optical flow, when circles become semicircles, we stop finding them.

However, one result of this is that the rectangles around the clocks would not be very stable. One goal we would have if we were going to continue to improve this project would be to better integrate the optical flow with the Hough Circles algorithm. We attempted this, but were most successful just letting it find the circles.

After evaluating the use of the Hough Circles algorithm for finding clocks, we found that it was extremely successful. First, it did not negatively affect the success of the other objects, and it only improved the clock. Also, it was between 0.7 and 0.8 F-Score for all clock environments that we tested, though at a visual inspection you would think 0.9 to 1 because it just didn't miss any clocks.

Optical Flow

Another one of our major extensions was to use optical flow for object tracking. To paraphrase the advice of professor Andrew Ng, we tried to get something simple working right away, and then iterate from there.

The first think we attempted with optical flow was to calculate a simple dx and dy velocity vectors over the whole image. We used openCV optical flow methods to do this. This produced relatively good results. It would return a dx and dy value ranging from 0.001 - 0.4 and we would multiply it by a scalar value and add it to the position of where we though the object was previously.

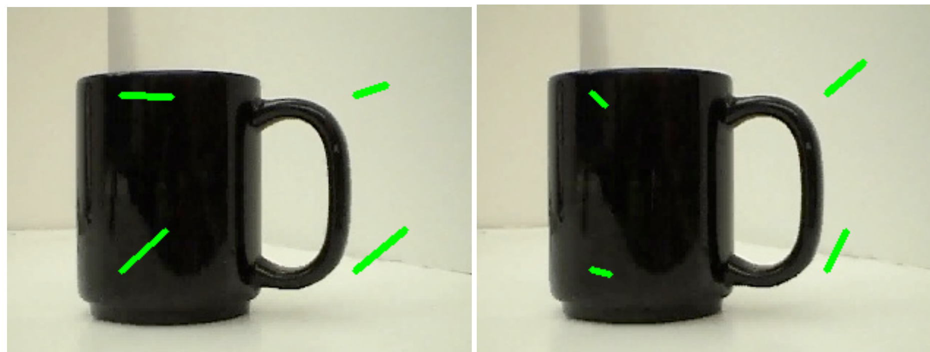
This was surprisingly successful right away, but even after multiple iterations, the improvement from optical flow was not that impressive. One problem with the average value that was returned was that it might have been rather noisy, so sometimes the image would move too much or too little depending on the video.

We tested optical flow scalar values from 5 to 100 to see how this affected our results. Overall, the optical flow had a small effect on our overall F-Score, mainly because we were having trouble integrating it in with the data we already had. We tried many methods to solve this problem that I will describe below. We settled on a value around 10 for a scalar of the optical flow dx and dy after testing it on several videos. One problem right away with this number and this model was that it was variable over different video environments. One place where this can be seen is in the video over the staplers and scissors where the camera is virtually still, but the optical flow average vectors adds up and shifts the objects off screen.

Another idea we contemplated in the use of optical flow was how often to use the flow information. In all of the videos we tested on (and presumably for most videos in general), there is not much movement between frames. Because of this assumption, you could either set some sort of maximum value for possible flow movement, or another idea we tested was to only run optical flow on some of the rounds. We tried running it every frame, every other frame, and every third frame. In all, the success rate differences were very small for all of these.

This was the first version of testing that we used. Then we tried to account for one potential downfall of only using one average flow vector over the whole image, which was the inability to distinguish between zoom and translation. In the ideal case we would hope that the average dx or dy vector would be zero for zoom because everything would "cancel out," but this hypothesis proved to be wrong.

The first thing I tried was to compute an average vector over the left side and right side of the image. I assumed if the camera was zooming then the dx for the left and right side would be of opposite sign, and if it was zooming it the dx for the left would be negative and the dx for the right would be positive and vice versa for zooming out. When we ran tests on this it would correctly identify sequences of frames that zoomed in and out considerably, but then we had to address the question of how to alter the object coordinates or object heights and widths with the knowledge that it was zooming. I tried several schemes: multiplying by some zoom factor (which proved to be a little to extreme), adding some zoom factor multiplied by the dx (to try and guess how the dx and dy vectors were related to the zoom amount), but these attempts were mostly educated guesses and failed to have a real positive impact on our viewable and F-score results.



Four Quadrant Optical Flow: The changes from frame to frame demonstrate its volatility

Then there was the possibility that this model was too simplistic so we tried to compute optical flow over the four quadrants of the image. In the screenshot above, you can see the variability between two consecutive frames (the screen shot is of the dx and dy flow vectors for each quadrant plotted in the center of the

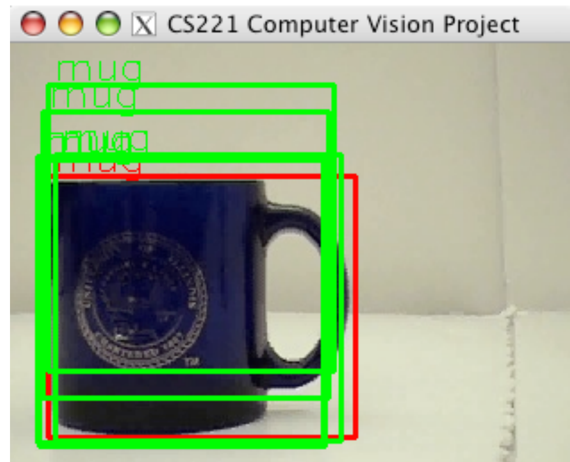
quadrant and scaled), and this goes to show how the camera was really moving so much and flow estimates were so volatile.

We thought about implementing optical flow over the four quadrants of the image, but we did not have time. We also thought about trying to get a median over the dx and dy matrix as opposed to the average to try and get rid of the noise.

However, overall, after many iterations of the optical flow algorithm there was little improvement from round to round, and also not a great effect on the F-score. We used other algorithms to supplement optical flow, mostly as a result of what we observed from the videos.

Rectangle Averaging and Non-Maximum Suppression

When we initially ran our logistic regression classifier we found that there were several obvious issues. One would be that if we found a mug we would end up labeling five mugs. You can se an example of this in the screenshot below.



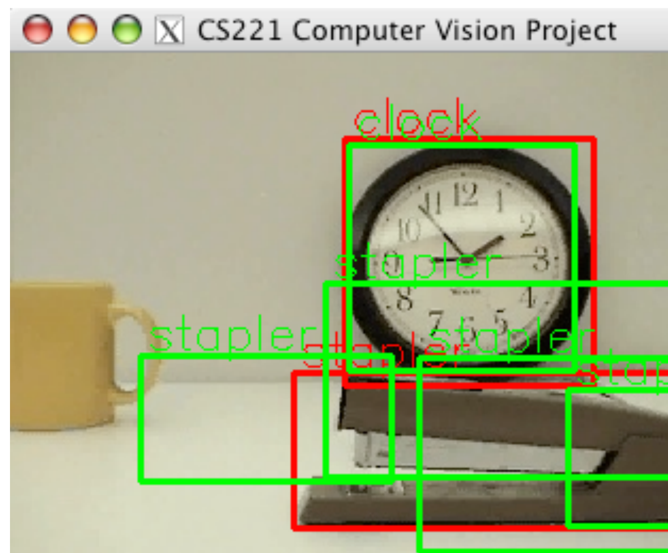
Without averaging, this yields numerous false positives

Our first intuition was to use non-maximum suppression, which we implemented to remove overlapping objects with a lower confidence. This provided a satisfactory solution, but after running more tests and watching the video, we thought of a better way we could approach this problem.

When you watch the videos you notice something very interesting, and that is that if you find a mug, you probably find a bunch of mugs, and the mug actually seems to be at the average location of all the mugs. When we used non-maximum suppression we were actually discarding all the information about the other objects we thought we found. However, we thought that this information was extremely relevant. First if we found 6 mugs all with at least 0.99 confidence, there really probably was a mug. However, if we found one stapler with 0.99 confidence, it is very likely that there was not a stapler. To simply use non-maximum suppression to get rid of those other five mugs would lose a lot of information. Also if we find 5 mugs with 0.99 confidence and 1 mug with 0.999 confidence that mug is not necessarily much better than the other mugs.

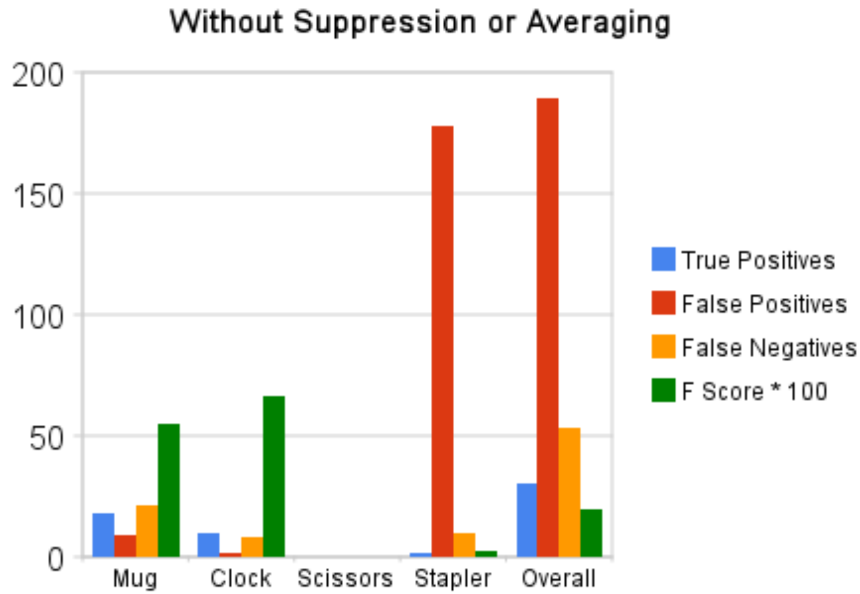
Our solution to this was to keep all the information by averaging all of the x and y coordinates of the same objects if they overlapped by a certain threshold as well as average their widths and heights. This caused a very noticeable improvement in our results. We contemplated doing a weighted average, but did not implement this. If we continued to work on the project we would consider this.

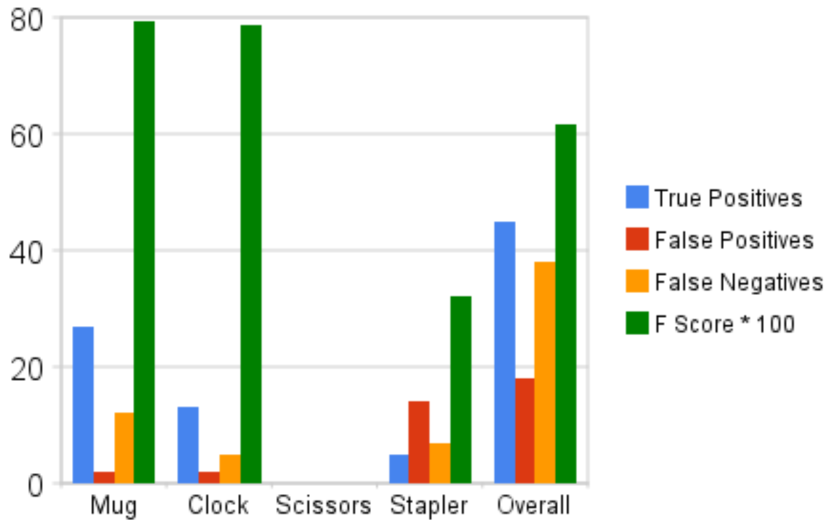
We also discovered late into the process how the overlap function in the CObject class worked. We wondered for several weeks why it would not get rid of objects that were overlapping by more than our threshold, but then upon examining the CObject class we found that overlap returned 0 if they were different objects. This gave us troubles when we would find a number of clocks, but also detect a mug on top of the clock. We modified the overlap function to handle the case of overlapping different objects, which allowed us to eliminate many false positives.



Lots of staplers that need to be merged

The Effect of Rectangle Averaging and Non-Maximum Suppresion





Clearly, the classifier performs much better with suppression, but more specifically with averaging and merging. Doing so cuts down on the number of false positives significantly. Without it, there were 189 false positives, while there were only 18 with the merging and averaging added. F Score is also significantly improved.

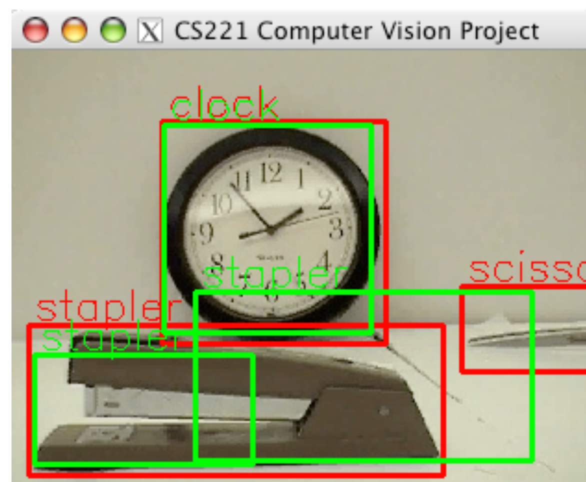
Our Specific Knowledge of Objects and Object Positions

There were several other post processing steps that we considered and implemented. We wrote a function to bring rectangles on the screen. Even if the object was partially off the screen, we thought the F-Score was computed by the size of the rectangle onscreen, so if we made that rectangle smaller, we could make the difference in area smaller.

We also wrote a function to remove rectangles that were completely off screen. We initially had a problem where we continued to draw boxes for objects that were off screen.

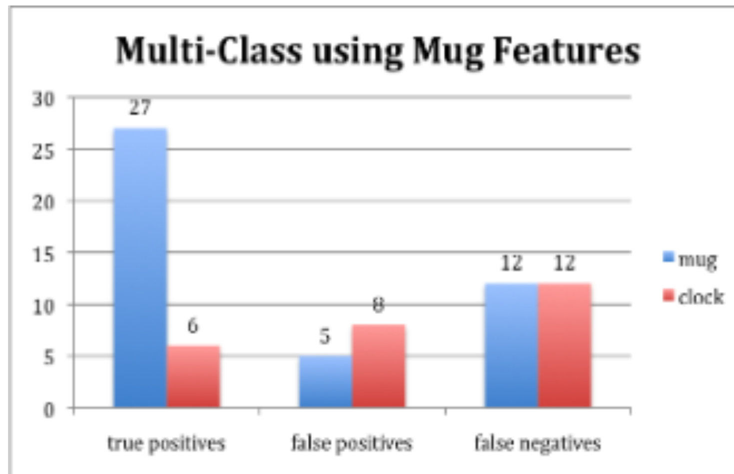
We also took care to write function that took into account our specific knowledge of the objects. Like with clocks, which we knew were circles, we knew certain things about the shapes of staplers and scissors and keyboards. Staplers are generally in the shape of a rectangle where the width is about twice as large as the height. There is also some analogous ratio for scissors and keyboards. We used this information to scale the proportions of certain objects to match what we knew these objects looked like in the real world. Scissors should never look like squares.

Also given the shapes of scissors and staplers, we modified the averaging algorithm. Rather than return the average x, y, width, and height, as we had done with mugs, we averaged the y and height, but took a range of x values. We were often finding a number of staplers next to each other of about the same height, but an incorrect width (too short). We merged them together to keep the average height, but put a box from the leftmost x to the rightmost x. As you can see, the two staplers it found in the image below need to be merged in this way.



Five-Class Decision Trees Using Mug Features

Since most objects don't have terribly distinguishing features we first trained all of our classifiers using only the fragment based mug features. We thought that reinforcing particular features on other objects might produce some interesting, diverse results.



The graph shows that mug features perform somewhat well on more than one class and are able to identify the clock. This first multi-class test yielded an F-score of 0.5, and it seemed that building a five-class feature dictionary would only improve this result.

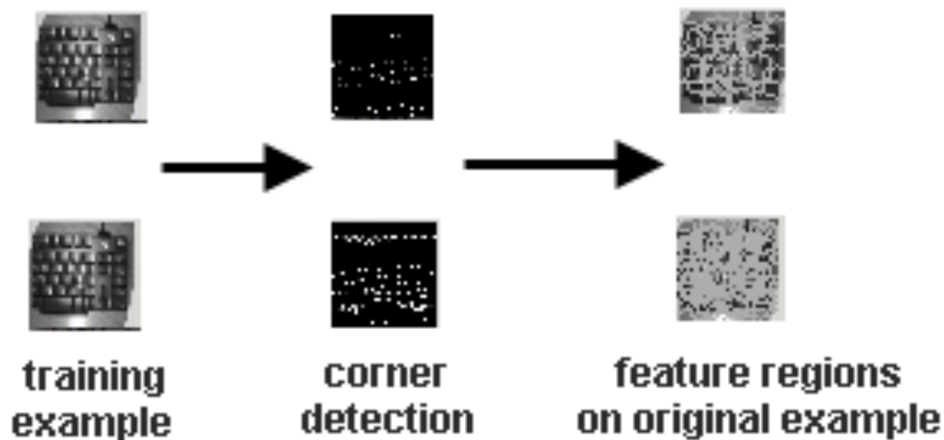
Fragment Based Features with Random Regions

Our first attempt at building a feature dictionary came by following the handout's specification for how the mug features were obtained. We extracted 10 features from each training example for roughly 15,000 features and attempted to train decision trees on an equivalent 15,000 training examples. To do this we wrote two extra binaries: ./makeDict and ./pruneFeatures. The make dictionary program extracts a specified number of features from each training example and builds an xml dictionary consistent with FeatureDictionary.h. ./pruneFeatures then trains decision trees on these features, determines which features were picked by the trees, and outputs those features to a new xml dictionary.

The problem with this approach –- using 15,000 features at once --- was that it was too slow. Training this way would have taken more than twelve hours and was not feasible since it would undoubtedly require some tweaking. Instead we tried to extract a smaller sample of fragments from each image and train on a smaller subset of the dataset. To do this we extracted two features from each example and trained on about 3000 images. We were able to successfully build a set of classifiers in a reasonable amount of time, but those classifiers found no objects in any frames.

Fragment Based Features with Corner Detection

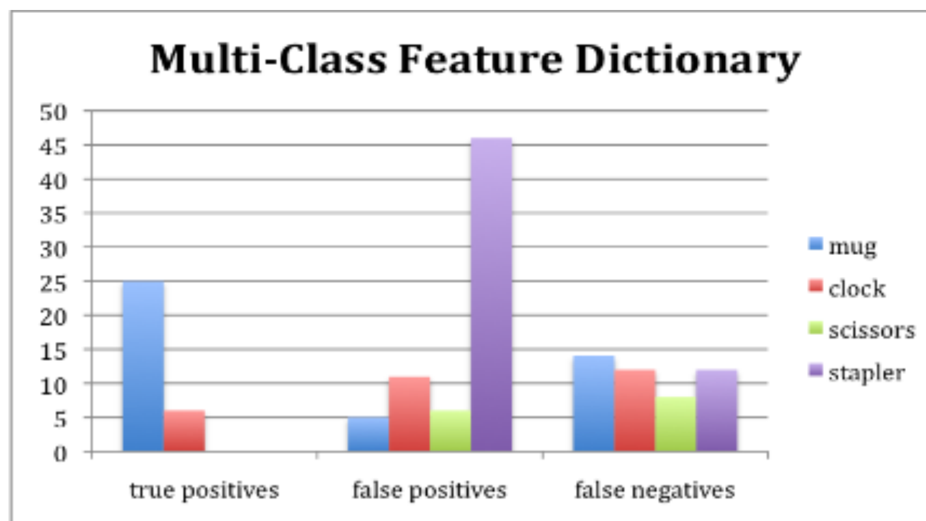
After trying random fragment regions and getting no worthwhile results in a useful amount of time, we decided that the five object classifier would need to start with fewer but more informative features. The mug fragment features were computed by training 3000 features on about 2300 training examples. A five-class system would thus need to train a total of 15,000 features on 15,000 training examples, as specified above, to be equivalently successful. Instead we decided to do region of interest scanning and extract fragment based features from those regions. We used corner detection to identify interesting regions in the image:



We mapped localized corners to regions in the original image and built a feature dictionary based on these regions.

To test the usefulness of this approach we first extracted one random region of interest from every tenth positive training example for roughly 300 features. We went straight to classification and trained our decision trees on these 300 features. Testing showed promising results. While random features had yielded no classification whatsoever, our initial region of interest tests classified objects about on the level of our first attempt at logistic regression.

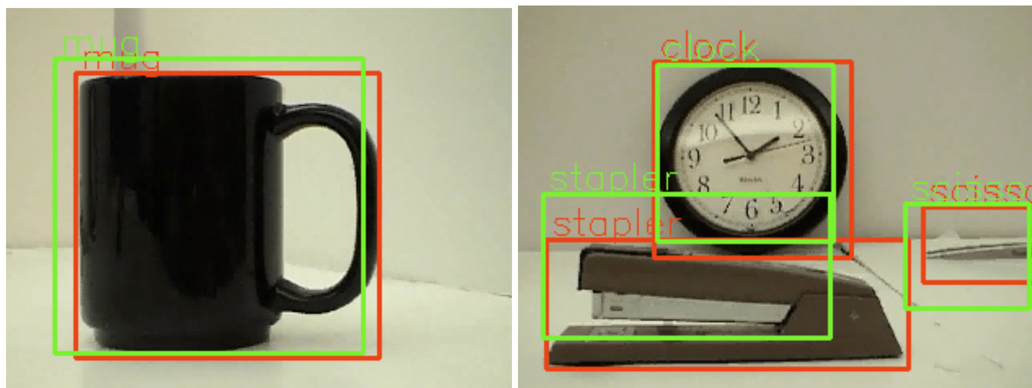
Convinced that corner features were a useful approach we pursued them further, this time extracting three features from each image and pruning our result down to approximately 200 features.



Our F-score declined dramatically. Our very first tests using only mug features had yielded an f-score of 0.5 while the shared feature dictionary could only muster 0.3. It found more of every class of object, but not well enough to score. We often found the general region of objects and labeled them, but the box area compared to the ground truth file did not have the right amount of overlap, marking the classifications as false positives.

Class-specific Feature Dictionaries

This weakness suggested two conclusions. First, while region of interest features worked well, a shared dictionary was not the right approach. Second, a robust method would need to use better test time techniques for distinguishing objects from their surroundings. We decided first to build class-specific dictionaries analogous to the mug's and try some experiments on these. We first successfully built a dictionary for the stapler and trained all classifiers on only the stapler dictionary as we had for the mug.



These classifications were done using a stapler feature dictionary only.

Interestingly, stapler features identified the mug better than they identified the stapler on the easy video. We concluded then that class-specific features were not that important, but distinguishing an object from its surroundings was utterly important. Mugs are easy objects to find. They often contrast with their surroundings and are square. Staplers, by contrast, are not square, and so squashing them into squares in the training examples and test time windows does not seem to be a useful approach.

A more robust approach would try a variety of window sizes at test time and use separate classifiers trained on class-specific feature dictionaries. Our very first attempt, examining only the mug using mug features ended up being our best test, and so utilizing five equivalent classifiers would help. Additionally, this approach would take advantage of different channels in the image as well as edge detection and other filtering methods to distinguishing the objects from their surroundings.

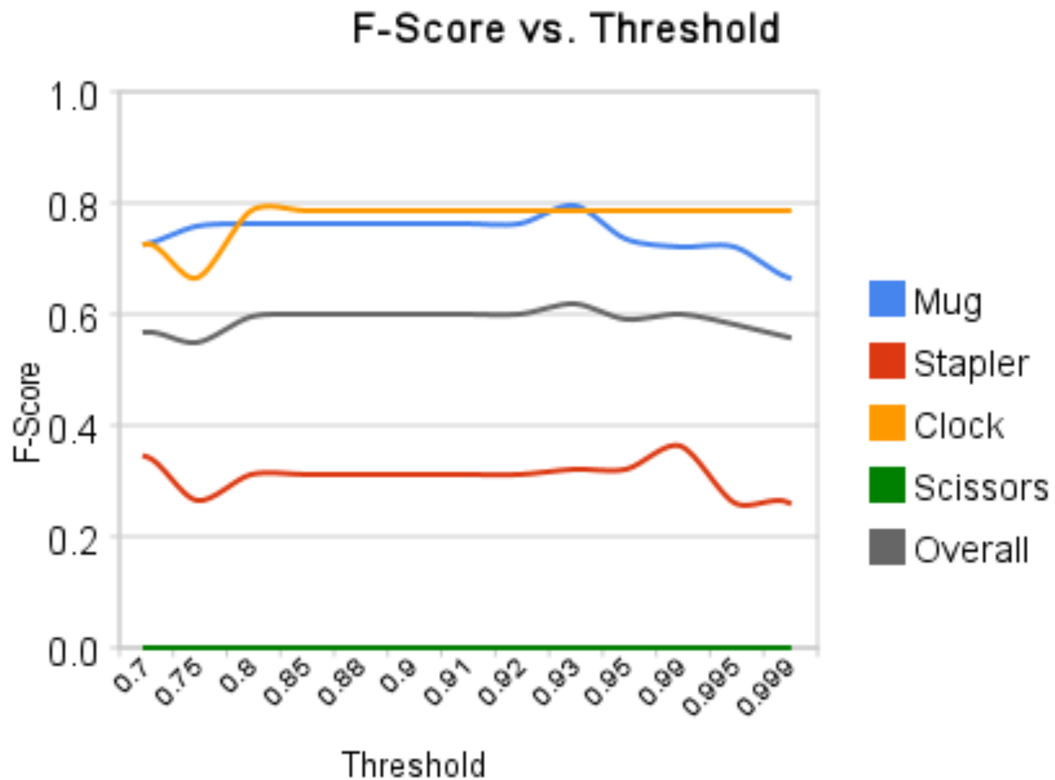
Miscellaneous Attempts

While working, we attempted several other extensions to see if they helped. After seeing the success of the Hough Transform for Circles, we implemented a Hough Lines function, but we couldn’t make use of the data.

We also created a function to view Canny Features on the video frames. We viewed the images, but we didn’t find a good way to integrate them with our classification.

Testing

One of the major efforts for our group was in testing. There were many complicated commands to run and many random directories to remember, so we created shell aliases and shell scripts to help automate testing. We modified test.cpp so that it could take in alternate flags. For example we added a -t flag, which took a double as an argument, and could use this to alter the threshold from the command line. In this way we could write automated scripts or aliases that would run multiple tests over different thresholds without having to re-make all the files. This was a very helpful procedure, and we could use this -t flag to adjust lots of other parameters in our model for better automated testing.



The graph above was run using the –t flag to test on a number of different thresholds on the easy video. Out overall F-Score peaked at .616 and stayed pretty constant over the high .50's. It dipped towards either end when the threshold was too low or too high, making classification too easy or too difficult, respectively.