Vision Project Report

Team Beta-Carotene

CS221 - Artificial Intelligence - Winter 2009

Stanford University

Team Beta-Carotene

Nikil Viswanathan – nikil@stanford.edu

Matt Bush – mattbush@stanford.edu

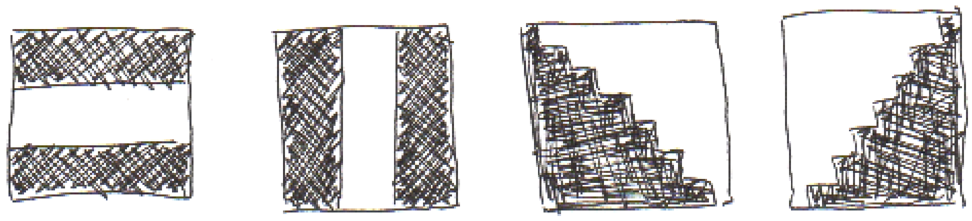
Riddhi Mittal – riddhi@stanford.edu

Table of Contents

1 Data Structures and Procedures

1.1 Haar Features

Our final classifier relies primarily on Haar Features, although some of the labels use additional features like correlation. The Haar Feature, and Haar Feature List, was an important data structure in our code. Each Haar Feature is represented as an instance of the Haar class, with a CvRect representing its rectangle, and an enum representing the feature type. We used the suggested 7 Haar Feature types (H, V, D, TL, TR, BL, BR), as well as 4 new ones (R, C, DD, DU) described below:



R **C** **DD** **DU**

R = Rows

C = Columns

DD = Diagonal, Down

DU = Diagonal, up

You'll notice that DD and DU are not straight diagonal lines, but are instead stair-like. That was because to compute the Haar Features most efficiently (especially when running multiple Haar Features on the same image), we precompute the Integral of the image and sample it at different points for each Haar Feature. Using the Integral to get the intensity at different points required us to use only rectangles to construct the shape of Haar features.

Haar Feature Lists are formatted with one Haar Feature per line—exactly the same as the file provided for us in the starter code. In the various stages of our program (feature generating, training, and running our classifier) we save and load Haar Feature lists very frequently!

1.2 TreeData

Both our Generate program and our Train program test lists of Haar Features against a large set of images, then they use a subset of those images to filter features, train decision trees, or set up CvBoost. To keep track of all the raw data produced by running images against Haar features in the early stages of Generate and Train, we designed a structure called TreeData.

TreeData is similar to the data that we pass into CvBoost in that it has a 2-dimensional matrix of feature variables (indexed by image number and Haar feature number), a 1-dimensional matrix of “labels” or whether each image is in the positive or negative class, and a 1-dimensional matrix of suggested Threshold values for each Haar Feature.

Because Haar Features and positive/negative labels per image differ between the five different labels (mug, stapler, scissors, clock, keyboard), we have one TreeData for each of the five labels. Each of the TreeData ultimately helps generate one Decision Tree or one CvBoost, as there is one of each per label.

1.3 Decision Trees and Boosted Decision Trees

In the milestone code, we wrote our own Decision Tree code in CClassifier. Each node in our Decision Tree chose the Haar Feature and Threshold with the highest information gain and split on it. Leaf nodes were nodes that had reached the maxDepth value or were homogeneously positive or negative. The decision tree wrote and read itself from files in a breadth-first fashion, and multiple decision trees were able to be written and read to the same file.

After the milestone, we decided to use CvBoost to get much stronger results. When implementing CvBoost we had to restructure some of our code in order to format our variables into CvMats suitable for CvBoost, but overall it thankfully didn’t affect our code structure that much. We replaced our custom save and load code with CvBoost’s built-in save and load functions. Since we have one CvBoost per label and thus multiple CvBoosts, each Boost writes itself to a different file: the provided filename with the name of the label it pertains to affixed to it.

1.4 CObjects and CObject Clusters

The most important data structure we used in Test (or CClassifier::run) was the CObject class provided for us. We did a variety of operations on the raw CObjects generated by our Decision tree or CvBoost, in order to filter them to get more accurate results.

For one, we clustered together nearby CObjects so that we wouldn't have redundant results. We used the CObject method overlap() as a quick way to figure out CObject similarity.

2 Extensions and Experimental Results

2.1 Generate: Choosing Haar Features

2.1.1 Rationale for writing a Generate program

The first step in our decision-making pipeline is choosing the Haar Features that will be considered when we build our decision tree from the set of all possible Haar Features. Since the original Haar Feature list given to us had only 57 features optimized for mug, we knew we had to come up with new feature lists, so we did so algorithmically. We came up with 5 different lists of effective Haar Features, one for each of the labels—mug, scissors, stapler, clock, keyboard. Initially we had 400,000 possible Haar features to start out with but then decided to increase the number of first images to test on from 12 to 24 so we had to go down to 150,000 Haar features to be able to compute efficiently on these images. We increased the width and height multiples from values of 2 to new values of 4.

To do this, we iterated through about 150,000 possible Haar features:

- all x and y values in multiples of 2
- all width and height values in multiples of 4 with a minimum of width + height ≥ 20
- the original 7 suggested Haar Feature types + 4 unique new types of Haar features (described above)

We tested these 150,000 features on a small number of images (24 from each label), recording the raw values that they gave each image. Then, for each label, we took the average values each Haar feature gave images that matched the label (positive group) and images that did not match (negative group) and used the halfway point between these 2 averages as that feature's threshold for classifying positive and negative for that particular label.

Using this method of classifying images, we chose the 5000 Haar Features that independently gave us the most information gain when classified to the small set of images.

2.1.2 Generating in Stages

We then narrowed down this set of 5,000 Haar Features to 1,500 Haar Features by testing them for information gain on a larger number of images (repeating the process above, but starting with 5000 Haar Features, and using a larger set of images), and then repeated again for a total of 5 stages. In each stage we began with a large number of features, ended with a smaller number passed on to the next stage, and tested with an increasing number of images. Our final result was 120 information-gain-optimized features for each label.

The generated Haar features were initially very redundant. There were many groups of Haar features that all looked at the same part of the image, and probably wouldn't be very useful as the split features in a decision tree (for example, the Haar Features "16 12 24 4 R" and "14 12 24 4 R" were the top two in the "stapler 10" list—they tell us essentially the same information and would make the same split). We changed the algorithm to test features to see how useful they'd be *given another feature*—that is, how much information gain we get from a feature classifying images that had already been split into two lists by another feature. This algorithm was $O(IF^2)$ rather than $O(F)$, since all features were tested against one another for this "dependent" information gain, and to compute information gain we need to consider the decision made on all images (hence the I term). Since the algorithm was quadratic with the number of features, we only used it with the 2nd through 5th stages, not the 1st stage (which would have had a considerable time cost). For the 1st stage we used our old $O(IF)$ algorithm; you'll notice this difference in the table below.

Stage	# of Images per label, at most *I*	# of Features considered *F*	# of Features ranked *R*	Running Time Formula	Running Time (Relative)
1	24	150,000	5,000	$O(IF)$	3,600,000
2	70	5,000	1,500	$O(IF^2)$	1,750,000,000
3	200	1,500	600	$O(IF^2)$	450,000,000
4	600	600	300	$O(IF^2)$	216,000,000
5	2,000	300	120	$O(IF^2)$	180,000,000

2.1.3 Ratios

We also varied the ratio of positive to negative images that we trained the Haar features on. We kept this image ratio when we trained both the trees that we created and the boosting trees. When we decided to add a "Ratio" aspect to our decision tree (see section 2.2.2 below), we generated a different list of features optimized for each ratio. This is why in the final submission, we have many lists of features (for example, the feature list for clocks with positive-negative ratio 1:10 is at "generated/clock10.txt").

2.2 Train: Training our CvBoost trees on Haar Features

2.2.1 Preparing Variables

Our method for training operates much like our method for choosing Haar Features, except the output is a fully structured decision tree (later a boosted tree) rather than just a list of useful features.

We ran the 120 Haar Features per label chosen by Generate (above) on the full training set of 40,000+ images and recorded the raw values that they outputted. We first iterated through images, then labels, then Haar features. Iterating through images first was the most efficient; we precompute the image's Integral and run all the Haar features on that integral, which entails extracting about 8-20 pixel values from the integral for each feature. (We originally were computing the Integral of the image every time we tested an individual Haar feature, which we learned was inefficient, so we restructured our code.)

2.2.2 Choosing images based on a Ratio

We then chose a subset of those images to include as training data for CVBoost. We didn't want to include all 40,000 images for all the classifiers. We originally did this and it was problematic, as the classifier was weaker for some labels than others. For example, the label "clock" has 40 images and the label "keyboard" has 400, thus 1 of every 100 images input into keyboard's Boost tree had a positive response and 1 of every 1000 images input into clock's Boost tree had a positive response. This discrepancy between 1:100 and 1:1000 made the tree output drastically different, so we decided to hold the ratio between positive responses and negative responses fixed.

To do this, we defined a constant BOOSTING_RATIO. After running our Haar features but before providing data to CvBoost, we selected images that weren't in the positive class (i.e. images that aren't a mug for the mug Boost tree) with a small probability based on BOOSTING_RATIO, such that the ratio of positive to negative-class images selected for CvBoost was approximately 1:BOOSTING_RATIO.

We also choose the Haar features that had been trained with this image ratio for the input. That way the decision tree and the Haar features had been trained on the same image ratio.

With these images selected, we then populated CVMats with the appropriate data and ran Boosting. Running Boost and saving the final completes the training process.

2.2.3 CvBoost Parameters

Finally, we tried different values for CvBoost's numSplits and numRounds parameters. We had been using 10-12 as the maximum depth of our decision tree before we started using CvBoost, so we initially used numSplits = 10, numRounds = 5. That proved to not be very effective—other classmates told use they used smaller numSplits(their trees were more like stumps), and more numRounds to make up for it. We tried different values and ultimately ended up using numSplits = 5, numRounds = 50.

2.3 Clustering: Grouping like observations

We went through several different iterations of clustering techniques. As we changed our decision tree and kept tweaking it, the clustering had to change to best fit the relative output of each of these tree iterations. We had some parameters which we could tune the clustering with but we also changed the basic structure of the algorithm as we went through different tree iterations.

2.3.1 Milestone

For the milestone we just implemented a simple clustering which took the maximum of each of the detected mug bounding boxes' coordinates. This was effective and simple and since we were using the easy video worked well, but there were limitations that we wanted to improve on. We limited the output to one object per frame with this technique.

2.3.2 Requiring Object Overlap

The next attempt that we took at clustering for general object types was to check if the object overlapped with any of the other objects with the same label. We threw away any object detection which did not overlap with a detection of the same object type. This helped cluster the decision tree output into regions of high probabilities. Thus this made it much more likely that we would actually have an object at the output detection position since it had many initial overlapping detections.

To implement this, we iterated over each of the frame's object detections and checked to make sure it overlapped with another object detection of the same object type. We discarded any frame that did not overlap with another object of the same type. We also had a variable threshold, which we tweaked, specifying how much shared detection box area was required to call multiple stacked detections an overlap. Then we averaged all of the bounding boxes of the resulting objects of the same type and clustered the output into one detection. We decided that it would be better to have this restriction in. This strategy was effective in reducing random object detections but we were checking every object detection with every other one so the clustering algorithm had an^2 running time and was very slow. We could run it with the classifier but when combining with other detectors we didn't think that it would be feasible.

We decided to limit the output results to have at most one object of each type per frame. We wanted to have cluster based on the location of the object so that we could bypass this limitation. However we decided to spend the time improving our decision tree classifier to have a steady output detection. The clustering was closely tied to the decision tree and we wanted to have a stable decision tree before we started to implement more clustering.

2.3.3 Average Object Overlap

An idea for improvement for the running time I had was to take the average of each of the object detections of the same labels and see if each which objects overlapped with this mean. We would keep the ones that intersected by some specified amount with the average and discard the rest. This improvement improved the n^2 running time to linear and allowed us to better focus the cluster on one area, since before there could be two major clusters in the required object overlap method that might be averaged.

We also came up with some ideas to implement a multi-cluster detection algorithm rand were going to implement it but realized that our time would be better spend fixing the decision trees. We decided to leave in the average clustering in the end for our final project. The method was simple yet still effective. Also this gave us time to run more advanced classifiers on the frame.

2.4 Correlation: A different feature approach

2.4.1 Intuition

We had the intuition that given an image I of the object O we want to detect in a frame F, convolving I with F should give us the brightest values at the points where image I of object O exists in the frame. We figured out the OpenCV function cvFilter2D and used that initially. Later however we just used the conv2 function from MATLAB to test lots of hypotheses quickly.

So we tried making templates by taking small distinctive patches of images we wanted to detect in another larger image, and convolved the template with the larger image. We did get bright patches at where the template was located in the larger image, but we also got a whole bunch of other bright images.

We created 10 templates to start with – 2 from each different object. We convolved each one separately with a frame image that contained the objects. We tried to detect patterns but nothing seemed to occur.

One consistent problem was that frame images that were already pretty bright because of the light in the frame, resulted in more bright patches in the convolved image. So we tried to make the template image intensities sum to zero. To do this we subtracted sum(template intensities) / (number of pixels in template) from the intensity of each pixel in the template. But this made the template image look very bad. And while there were some cases where this seemed to help, there were others where this made detection much worse.

We also tried making the frame image intensities add up to zero in a similar fashion, but did not get much out of it.

One of us discussed the idea with Ian, and he emailed us some suggestions. His first suggestion was that we pre-flip the patch (because the convolution function flips the kernel). His second suggestion was that we compensate for the fact that the convolution is going to be brighter when the image is brighter, so we should subtract out the part of the image that the kernel covers. We tried his second suggestion and subtracted out the mean of the part of the image the kernel / template covered. While we felt that the resulting convolved images did probably get better, there was still no pattern to the brightness patches. They were all over the place.

After doing research on links Ian gave us and on Wikipedia, we saw that it was Normalized Cross Correlation, not convolution, that we should work on. Now, we needed to figure out the OpenCV function that would do this for us. The document at http://www.seas.upenn.edu/~bensapp/opencvdocs/ref/opencvref_cv.htm#cv_imgproc_filters helped a lot. We found out that cvMatchTemplate has a couple different ways of performing the kind of template image matching we desired.

We ran tests with all six method types. Out of the six, three seemed to work nicely – CV_TM_SQDIFF_NORMED, CV_TM_CCORR_NORMED, CV_TM_CCOEFF_NORMED. We decided to choose CV_TM_CCOEFF_NORMED as the method we would use. Using this, we now needed to find the maximum intensity pixels in the image. We used cvMinMaxLoc to accomplish this.

2.4.2 Data Structures and Implementation

We now needed a framework to actually collect good image templates, and then load and test them on the frame images.

The data structures we created are as follows:

```
vector<string> ScissorTemplates;

vector<float> ScissorTempThresh;

vector<CvRect> ScissorRect;

vector<string> ClockTemplates;

vector<float> ClockTempThresh;

vector<CvRect> ClockRect;
```

```
vector<string> KeyboardTemplates;

vector<float> KeyboardTempThresh;

vector<CvRect> KeyboardRect;
```

and similarly for Mugs and Staplers (although we never used them for the final submission because they did not work too well).

For example, the Keyboard Detector:

- ⇨ KeyboardTemplates is a vector of strings that stores the names of the image template files to be loaded and correlated with the frame images.
- ⇨ KeyboardTempThresh is a vector of floats that stores the threshold above which the correlation of the image template in KeyboardTemplates at that index, with the frame image, would be considered a positive i.e. would correspond to a keyboard.
- ⇨ KeyboardRect is a vector of CvRects that stores the relative dimensions of a CvRect for the object detected, for the image template at that index.

A lot of image patches were tried as templates on the four videos available to us. We chose a few templates for each object that seemed the best – they were general enough to detect the object in most of the videos.

Next, we tried to minimize the number of false positives detected. So we ran tests with our template images and printed out the correlation number. We chose the correlation threshold to be a number that was high enough to minimize false positives.

Next, we set CvRects for the template features we were detecting. Since we knew what part of the object each template feature was, and what part of the object it would be detected at, we set what would be the CvRect around the point of maximum brightness, if the image correlation was positive.

So, for each frame, we match it with all the templates of all object types we have. We compare the correlation number to our threshold and determine if the correlation is positive. If yes, we look up the corresponding rectangle that should be drawn on the frame relative to the correlation point i.e. point of brightest intensity in the correlated resultant image.

Even with only about four or five templates per object, the correlation worked pretty well for the Keyboard and the Clock, followed by the Scissor. If we had added many more templates, it would work even better. We did not see it working very well for the Mug, except that it worked for the mug's handle. But the templates also confused the mug handles with the right hand side of the clock. We did not see the Stapler too promising either. So we decided to only let the Keyboard, Clock and Scissor detectors remain.

2.5 Optical Flow: Choosing output based on motion

We decided to implement optical flow to help track the objects from one frame to the next and help eliminate frames. We initially wanted to use this to filter out frames which didn't move in the same direction as the flow and then decided to use it to add frames in based on the probability of the previous frame detections. On the process to our final implementation, we went through several design iterations of optical flow. The major design tradeoffs we were focusing on the frame processing time limit. We were concerned with the running time versus the accuracy of optical flow for each specific object.

2.5.1 1st Implementation

First we used a basic Lucas and Kanade optical flow algorithm which computed the flow for every pixel of the image. Initially this seemed like the best method, as we were getting the flow for every point in the frame, which meant that we could get the flow for every object, but then we realized this was excessive. The reason was that nearly all of the objects in the video were stationary so that the flow for the frame was the same as the flow for each object. The exception was when the camera zoomed in or out. Thus we just averaged the flow from each pixel to get the flow for the whole frame.

2.5.2 2nd Implementation

Since we were just averaging all of the flow vectors of all of the pixels of the first frame, and the objects were mostly stationary, we decided that we didn't need to calculate the flow for every single pixel in the frame. We decided to use another implementation of flow which would be more efficient given our assumptions. So, we moved to the sparse optical flow calculated by the iterative Lucas-Kanade methods in pyramids. We based this optical flow implementation on the notes by David Stavens (TA for CS 223B) who explained a method to do this. We used the OpenCV sparse flow function to calculate the optical flow and to calculate the points to track in the flow we used a function which found the strong corners in the frame using the Shi and Tomasi method. This way we didn't have to calculate optical flow for every pixel in the frame. Also since we had most of the objects staying in the same position relative to each other, this was not a problem and since we had multiple points we could even figure out zooming.

2.5.3 3rd Implementation

Instead of using the Shi and Tomasi strong corner detection method we decided to just choose parts of the strong clustered objects in the previous frame to track. With this we realized that we can pass in the parts of objects that we wanted to track specifically and use these as the points for calculating optical flow on. This method is much more efficient thank the first method and more flexible than the second since we only calculate flow on the points that we actually need to monitor. This also allows us to use either the observations from before or the final output from the previous frame.

In the end we disabled optical flow for the final submission since we wanted to make sure that it wouldn't actually perpetuate a false positive. We hoped to get a strong confidence on a tree before we activated the optical flow so that it would follow correct observation. In the end the optical flow computed quite quickly, especially compared to the clustering, and the flow vectors were relatively small since the camera didn't move quickly. Thus even though the flow wasn't enabled in the final project, it definitely was a learning experience and was interesting to see how we could actually apply the optical flow algorithm to the image tracking problem.

3 Final Performance: Strengths and Weaknesses

3.1 Final Decisions

Finally, we decided after many experiments and tests that we were going to use the boosted decision trees with the custom Haar features, the integrated correlation classifier and detector, and the clustering. We fed the milestone Haar features for the mug classifier, instead of the features that we generated, into the boosted decision tree (more explanation in 3.2 experimentation). We also weighted the final keyboard and scissors correlation detector to count as 10 individual frames of the decision tree detection system. Since we trained the correlation classifier to have very few false positives, we assumed that if it classifies something then the classification it is most likely correct. Also the clock correlation classifier was very good but it had some false positives, so after multiple tests we decided not to boost its weight and just count it as one object detection. We used the quick average clustering method in the end. We left on all of the object classifiers from the boosted decision trees.

3.2 Experimentation

We carefully experimented with the parameters for the Haar feature decision trees for both our trees and the boosted ones. We carefully tweaked the image ratios and the maximum tree depth cut off to see which trees would give us the best output. Surprisingly we found out that the provided mug Haar features for the milestone performed exceptionally well compared to the mug Haar features that we had trained. Also the various decision tree and Haar feature training image ratios gave very distinct numbers of object detections.

Without boosting ratio 50 and depth 10 worked best.

With boosting using 50 weak learners / depth 5

Ratio	Number of Output Classifications
10	Very High
50	High
100	Medium
120	Low

We found that the boosted tree using the 1:100 ratio gave us just enough output classifications to find the object but didn't overwhelm the clustering. With the 1:10 and the 1:50 ratio, the entire frame was covered in detections. With the 1:120 ratio, we didn't have enough detections to cluster and still cover the actual object.

We also found out that we were testing these parameters on a separate boosting file with the mug features. When we actually consolidated the code with the whole project and implemented the boosted tree detection for all of the object types, for some reason there were very few object detections. We found out that the images were sampled in a different way in the main code versus our milestone code, in the full project we took each image from the object and other categories with probably 1/ratio but with the milestone code we took the images from all the other object types first and then took the rest of the images necessary to have the correct positive to negative ratio from the others category. However even with changing this method we still didn't have combined classifier outputting enough frames.

To counter this we relied heavily on the correlation factors for the clock, keyboard, and scissors. We boosted the frame observation count of each keyboard and scissors detection from correlation. The clock was accurately detected by the added correlation features, especially when it found all three features, so we focused on this more. Ideally we would have tweaked the decision tree and set a better threshold for score cutoff determining the boundary of classification between positive and negative samples. However under the time crunch we had to make a strategy decision and use the correlation instead. Ideally, we would love to have a decision tree that actually outputted more detections and especially one that we could rely on completely for our extensions.

3.3 Final Results

We implemented several extensions but couldn't take advantage of many of them since our decision tree was unpredictable. In hindsight, we should have established our decision tree earlier and then build extensions off of it. This way we would have been able tune to the extensions better and have been able to produce a much better detector. Also this would have given us time to play around with interesting ideas such as only using the decision tree for the mug and anything else it was good at and using the other classifiers such as correlation for detecting the other objects.

3.4 Conclusion

This was a great learning experience in both science and team strategy. We will definitely try to get the core functionality and algorithm working very strongly in the beginning before we add the extensions. Also we will definitely start working on the project much, much earlier. All in all, we had a great time and learned a lot while working on this exciting project! We also developed a great appreciation for the amazing capabilities of the human body!