CS221 Vision Project Report

Team: Cognitive Dissonance - Alec Go, Elizabeth Lingg, Anand Madhavan

March 20, 2009

1 Description of the data structures and code

For the most part, our code is organized such that if there is a class, there is usually a header (.h) and source file (.cpp) associated with it by that name. In cases where it is not, the class may have been combined into a single file and in such cases, we explicitly mention the file locations. We organize the descriptions based on various functionalities.

1.1 Classifier

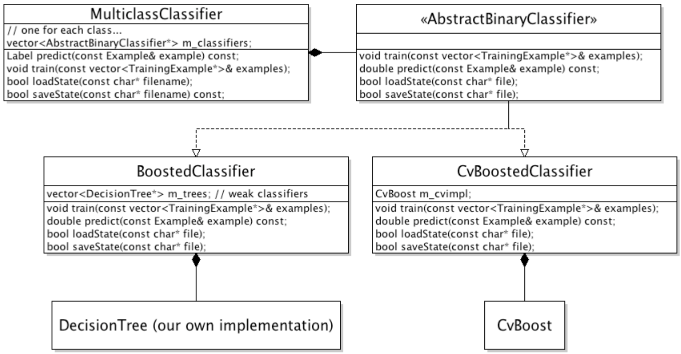
MulticlassClassifier
// one for each class...
vector<AbstractBinaryClassifier*> m_classifiers;
Label predict(const Example& example) const;
void train(const vector<TrainingExample*>& examples);
bool loadState(const char* filename);
bool saveState(const char* filename) const;
«AbstractBinaryClassifier»
void train(const vector<TrainingExample*>& examples);
double predict(const Example& example) const;
bool loadState(const char* file);
bool saveState(const char* file);
BoostedClassifier
vector<DecisionTree*> m_trees; // weak classifiers
void train(const vector<TrainingExample*>& examples);
double predict(const Example& example) const;
bool loadState(const char* file);
bool saveState(const char* file);
CvBoostedClassifier
CvBoost m_cvimpl;
void train(const vector<TrainingExample*>& examples);
double predict(const Example& example) const;
bool loadState(const char* file);
bool saveState(const char* file);
DecisionTree (our own implementation)
CvBoost

Figure 1: Class diagram for our multi-class classifier

Our classifier code uses a bit of object oriented mechanisms to enable quick testing with different implementations. This is described below:

classifier.h/.cpp: Modified code from initial template. Contains the class *AbstractMulticlassClassifier*, which currently has only one implementation in *MulticlassClassifier* described below. The idea behind the *AbstractMulticlassClassifier* interface was to be able to extend this to SVMs and other classifiers at a later stage if required.

MulticlassClassifier.h: This is our main multiclass classifier. This class contains one *AbstractBinaryClassifier* per class (eg: one for mug, one for keyboard etc). The underlying implementation of the *AbstractBinaryClassifier* interface can be one of either *BoostedClassifier* or *CvBoostedClassifier* (which uses *CvBoost* under the hood). This is represented as a class diagram in Figure 1. The depth of a tree and the number of trees are specifyable through constructor arguments to *MulticlassClassifier*.

BoostedClassifier.h: This is our own 'homegrown' implementation based on AdaBoost. It implements the *AbstractBinaryClassifier* interface and further uses class called *DecisionTree* which is our implementation of a simple decision tree that performs binary classification.

CvBoostedClassifier.h: This class also implements the *AbstractBinaryClassifier* interface and uses instead the *CvBoost* implementation of OpenCV.

DecisionTree.h: Our implementation of a simple decision tree that performs binary classification. It uses a n-ary recursive tree structure to store information. It also uses weights for examples, so the sampling distribution can be changed efficiently.

Label.h: Labels for the classifier are specified here. *Label* is currently a typedef to unsigned int. This file also contains functions that map the unsigned int to string and vice versa for efficient usage elsewhere.

NOTE: The performance of our *BoostedClassifier* is comparable to *CvBoost* in precision, recall and other measures (more details in Section 4.2) . However, our final submission version uses *CvBoost* by default, since it is considerably faster than our *BoostedClassifier* implementation.

1.2 Feature Extraction

HaarFeatures.cpp: This class allows us to extract haar features for a given frame.

EdgeDetectionFeatures.cpp: This class allows us to extract sobel (edge detection) features for a given frame.

Hough.cpp: This class allows us to extract hough, canary, and harris (circle, line, edge, and c orner) features for a given frame.

1.3 Motion tracking

TODO (Alec to fill in)

1.4 Cross validation tool

We introduce a cross validation tool as described in Section 2.1. This can be built using ‘make tune’.

tune.cpp: Main code for doing k-fold cross validation (described in more detail in Section 2.1). Reuses the *CClassifier* code.

Stats.h: Utility class for storing and computing statistics of precision, recall and F1 scores.

1.5 Infrastructure code

Other files contain infrastructure utilities for rapid development:

Timer.h: A utility class for printing out timing information for functions.

CommandOptions.h: A utility for processing command line arguments.

FinalSettings.h: This is a one stop to override all the command line options since we want *test* and *train* executables to work with the final fixed values of these settings for final submission (and not rely on command line inputs).

2 Implemented Extensions

We implemented a number of useful extensions. Some of these were just diagnosis tools and classifiers, others were features for specific objects and some others were for motion detection.

2.1 k-Fold cross validation tool

We implement a utility to perform k-fold cross validation of our features and classifier. The utility (called ’tune’) takes various command line options, such as size of the tree to be used and depth of tree to be used and performs a k-fold cross validation using examples chosen at random.

We go through all the training examples and shuffle them up. This gives us a uniform distribution of samples. We then use the first ‘n’ examples (specified by a command line argument) and perform k-fold cross validation on it. The first 'fold' is chosen as test set and trained on the rest of the folds. Then the second fold is chosen as the test set and trained on the rest of the folds and so on for all folds. We report the average test as well as the average training error. We report the precision and recall for each category. Finally we also report the confusion matrix. Many other parameters can be specified as command line options as well, these are documented in the Appendix 6.1.

2.2 AdaBoost decision tree implementation

We implement a fully functional AdaBoost[2] based boosted decision tree (as class *BoostedClassifier*). This class uses exponential weighting of falsely predicted examples and the weights are passed along to the nodes in a simple decision tree classifier. The tolerance is picked based on the average feature values encountered. We present results of using our classifier in Section 4.2. We note that although this classifier performs quite well, it is quite slow in comparison to the CvBoost implementation. For the final submission we choose instead to use the CvBoost implementation for its superior speed.

2.3 Hough based features

2.4 Histogram of gradients based features

TODO (Liz to fill in)

2.5 Kalman Filter

TODO (Alec to fill in)

We chose to implement the Kalman Filter to help reduce the noise of our image recognition algorithm. The algorithm has two primary steps: - Update - Predict

The major limitation with the Kalman Filter is that it assumes:

1. linear dynamics
2. the current state depends on the immediate past state (and not all past states).

Thus, it was very important to choose the correct blob to track.

2.6 Lucas Kanade

TODO (Alec to fill in)

We chose to implement the Lucas Kanade algorithm. The test videos met the three assumptions required by this algorithm [1]:

1. Brightness Constancy. We assume that in the test videos, lighting will be consistent.
2. Temporal persistence. The test videos had steady movement. Only the camera moved. The objects in the scene never moved.
3. Spatial coherence

Because the test video fulfilled these three assumptions, we thought the algorithm would perform well.

Talk about how you need to consider objects entering frame. Kalman filter doesn't have this problem. Discuss using the mean of the points - wasn't a good idea because it would skew towards many points. Talk about bounding box. False positives kill optical flow algorithm.

2.7 Lucas Kanade based interpolation

TODO (Alec to fill in)

3 Assumptions

TODO Not sure what we assumed here: listing hypotheticals: We assumed that the video does not change too rapidly. We assume that input trained images were all in grayscale. This was however also verified to be true.

4 Experimental results

We ran a number of experiments on each of our extensions. We present the results of our various experiments below.

4.1 Baseline classifier

We perform basic analysis of the CvBoost decision tree. We use CvBoost::GENTLE type with a fixed split criteria of 0.5. We use our k-fold cross validation tool to tweak the size of the trees used, as well as the depth of the trees used. We also use our tool to arrive at a 'bang for the buck' set of parameters, so as to minimize development time while still giving good enough accuracies. We use a 4-fold cross validation for all the below experiments. Average test and training errors are reported across these folds. In analyzing our baseline performance, we use only the 57 Haar features from the milestone run.

4.1.1 Effect of maximum depth of tree

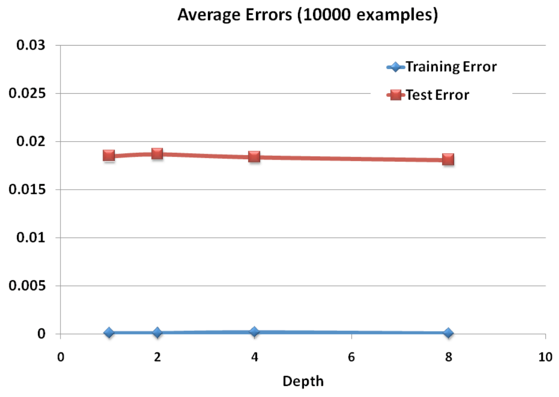

Figure 2: Baseline classifier: Effect of increasing depths on error.

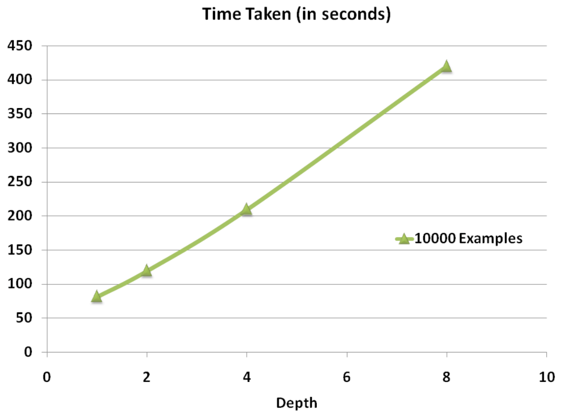

Figure 3: Baseline classifier: Time taken with increasing depth of trees.

Using 10000 examples, we notice that varying depths of the tree, we get better accuracies with increasing depths. However we also note that the time taken increases almost disproportionately for the improvements in accuracies obtained (see Figures 2 and 3). For example for depth 1, we see an accuracy of 1.85% accuracy, while for depth 8, this improves to about 1.81%. However given the time it takes (81 seconds vs 419 seconds), we can judge that increasing the depth a practical way of running our development cycle. Thus we arrive at an optimal depth of 1 or 2.

4.1.2 Effect of number of trees used in boosting

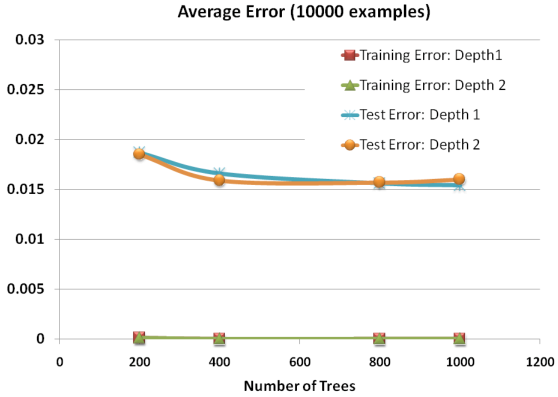

Figure 4: Baseline classifier: Effect of number of trees on errors.

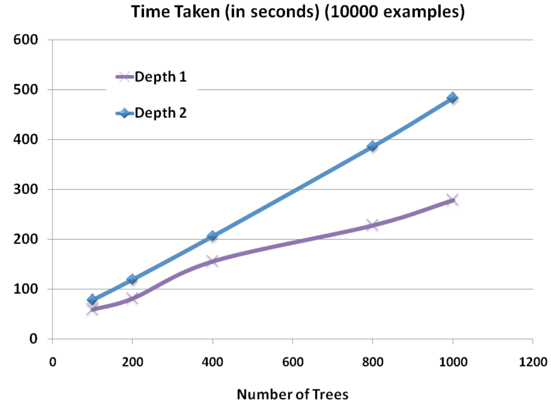

Figure 5: Baseline classifier: Time taken with increasing number of trees.

We next perform experiments by varying the number of trees used during boosting in CvBoost. With increasing number of trees we see that we get lower and lower training errors, but also lower and lower test errors upto a point, after which the results seem to plateau (see Figure 4). Doing a time analysis, we notice depths of 1 perform much better on large number of trees (see Figure 5). This makes us pick depth of 1 and use 400 trees for our development. However, note that we use larger number of trees (1000), and depth 2 for our final submission.

4.1.3 Effect of number of training examples

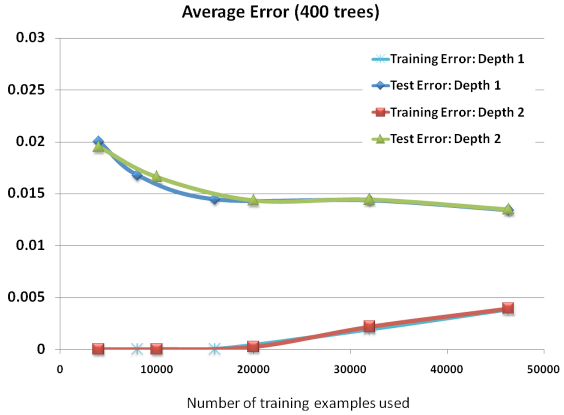

Figure 6: Baseline classifier: Effect of number of examples on error

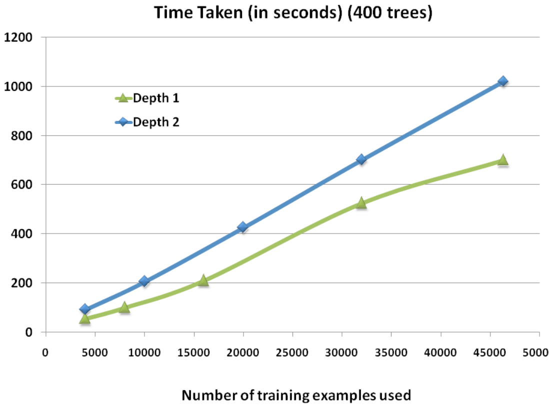

Figure 7: Baseline classifier: Time taken with increasing number of training examples

We perform a basic analysis of the effect of the number of examples on the test error. We hope that this gives us insights into how many examples we should use for our development. Here we notice that our curves follow the typical training and testing error curves, with gradually increasing training errors an decreasing test error. Test errors are typically around 1.5% compared to training errors which vary from 0 to 0.3%. At the least this validates our model as not having high bias. Also while there is an obvious benefit from using more data, we also notice that for 16000 examples, we get test errors of around 1.44%. This just takes around 210 seconds, while using all the 46359 examples gives us a test error of 1.33% but takes 699 seconds (an increase of 3 times). From Figure 6, we also notice that the 'knee' of the curve maybe occuring with 16000 examples. We thus use this for development. Of course we use all the data for training our classifier for final submission.

4.2 ‘Homegrown’ Adaboost classifier

Extending our work from the milestone, we also designed a multiclass classifier that implements the AdaBoost algortihm for boosting (in class called *BoostedClassifier*).

	‘Homegrown’ version	CvBoost version
Average training error	0.00102101	0.00386118
Average test error	0.0169115	0.0133954

Table 1: Comparison of average test and training errors using the two implementations.

The errors compared to the *CvBoost* classifier are shown in Table 1.

‘Homegrown’ version				CvBoost version			
	Precision	Recall	F1 score		Precision	Recall	F1 score
mug	0.705	0.617	0.658	mug	0.819	0.597	0.690
stapler	0.786	0.672	0.724	stapler	0.828	0.578	0.681
keyboard	0.633	0.638	0.636	keyboard	0.830	0.519	0.639
clock	0.313	0.939	0.469	clock	0.885	0.469	0.613
scissors	0.693	0.779	0.733	scissors	0.900	0.648	0.754
other	0.992	0.991	0.992	other	0.989	0.989	0.994

Table 2: Comparison of precision, recall and F1 scores for the two implementations.

The precision and recall numbers for the two classifiers are shown side-by-side in Table 2. Of course, the two methods use different variations of boosting, so the numbers are only used primarily as a validation of our classifier's performance.

4.3 Object detection features

4.3.1 Hough Transforms (Circle and Line Detection)

Without circle detection				With circle detection			
	Precision	Recall	F1 score		Precision	Recall	F1 score
mug	0.929	0.553	0.693	mug	0.947	0.574	0.715
stapler	0.805	0.522	0.633	stapler	0.793	0.515	0.624
keyboard	0.833	0.489	0.616	keyboard	0.929	0.707	0.802
clock	1.000	0.400	0.571	clock	1.000	0.400	0.571
scissors	0.938	0.571	0.710	scissors	0.953	0.581	0.722
other	0.987	0.998	0.993	other	0.989	0.999	0.994

Table 3: Comparison of precision, recall and F1 scores with and without hough circle detection.

Hough transforms are features that are used for shape detection in order to do object recognition. In this case, we focused on circles and lines within images. Circles were used for detecting clocks and lines were used for detecting all five objects. Before performing the Hough circle transform, we smoothed the image with a Gaussian.

We found that circles were a useful feature, allowing us to avoid misclassification and false negatives, most notably in keyboards. We tried several circle features such as highlighting circles in the image and performing a histogram on the image based on pixel intensity. We also used the number of circles in the image as a feature, which turned out to be our best feature. The performance increase using 4 fold cross validation is shown in Table 3.

We found that the Hough line features did not increase performance. This was partially due to background noise in the images as well as the imprecision in the line detection algorithm and the fact that the algorithm produces lines of infinite length (instead of line segments). Some of the features we considered were the number of lines in the image, as well as highlighting the lines in the image and performing a histogram based on pixel intensity. Future work would be to analyze different line features.

4.3.2 Edge Detection features

Without line detection				With line detection			
	Precision	Recall	F1 score		Precision	Recall	F1 score
mug	0.929	0.553	0.693	mug	0.929	0.553	0.693
stapler	0.805	0.522	0.633	stapler	0.847	0.537	0.658
keyboard	0.833	0.489	0.616	keyboard	0.947	0.587	0.725
clock	1.000	0.400	0.571	clock	1.000	0.400	0.571
scissors	0.938	0.571	0.710	scissors	0.909	0.571	0.702
other	0.987	0.998	0.993	other	0.988	0.999	0.993

Table 4: Comparison of precision, recall and F1 scores with and without edge detection.

After running Hough transforms, we considered that instead of analyzing the frames for specific shapes, we could use a rough outline of the objects in the image as a feature. Using the Canny edge detector, we were able to create an outline of the object. We then converted the outline from grayscale into a black and white image. The edges of the objects were highlighted in black, with the background in white. We created a histogram feature with two bins, one containing the white pixels, the other containing black pixels.

We found that this was an effective feature overall, increasing performance for staplers and keyboards (see Table 4.

4.3.3 Corner Detection features

Without corner detection				With corner detection			
	Precision	Recall	F1 score		Precision	Recall	F1 score
mug	0.929	0.553	0.693	mug	0.898	0.564	0.693
stapler	0.805	0.522	0.633	stapler	0.796	0.552	0.652
keyboard	0.833	0.489	0.616	keyboard	0.908	0.750	0.821
clock	1.000	0.400	0.571	clock	1.000	0.400	0.571
scissors	0.938	0.571	0.710	scissors	0.969	0.600	0.741
other	0.987	0.998	0.993	other	0.989	0.998	0.994

Table 5: Comparison of precision, recall and F1 scores with and without corner detection.

We guess that our classification of keyboards could benefit from using corner detection algorithms, since there are many corners on the keyboard. We used the Harris Corner Detection algorithm in the cvFindGoodFeatures function to detect the number of corners on an object. The number of corners as such, were used as a feature, and increased performance for staplers and scissors. Most notably however, we found a large performance improvement for keyboards, confirming our initial hypothesis. The results are shown in Table 5.

4.3.4 Histogram of Oriented Gradients (HOG)

Without HOG				With HOG			
	Precision	Recall	F1 score		Precision	Recall	F1 score
mug	0.929	0.553	0.693	mug	0.952	0.638	0.764
stapler	0.805	0.522	0.633	stapler	0.881	0.664	0.757
keyboard	0.833	0.489	0.616	keyboard	0.903	0.707	0.793
clock	1.000	0.400	0.571	clock	1.000	0.400	0.571
scissors	0.938	0.571	0.710	scissors	0.962	0.714	0.820
other	0.987	0.998	0.993	other	0.992	1.000	0.996

Table 6: Comparison of precision, recall and F1 scores with and without HOG.

We used a histogram feature on the gray scale image, using a histogram of oriented gradients as the theoretical foundation. HOG involves three stages, gradient computation, orientation binning, and block normalization.

First, we implemented stage one, by computing the gradient of the image, using a sobel operator. Then we implemented stage two, orientation binding, by creating the histogram of the object based on the image intensity values. Finally, we implemented stage three, by normalizing the histogram.

After some experimentation, we found that the feature worked best on the original gray scale image, without using the sobel operatior. With the sobel operator, we did not achieve a significant performance increase. Further work would include using different operators to compute the gradient of the image.

The results in Table 6 is with histogram normalization and 80 bins (without the sobel operator). We found that we got the same results with or without histogram normalization. Figure 8 shows the effect of number of bins on the F1 scores and we notice that scores improve progressively for clocks with higher bin sizes, while not playing a role in the others.

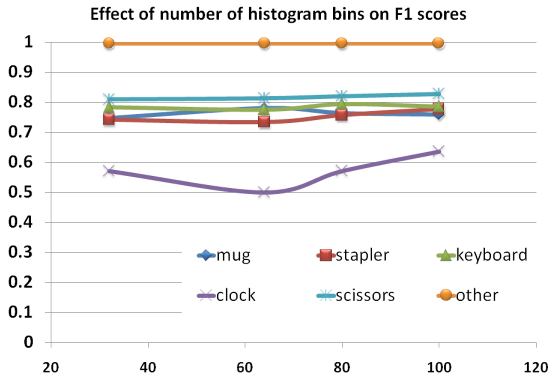

Figure 8: Baseline classifier: Effect of increasing depths on error.

4.4 Effect of coalescing of rectangles during motion tracking

No coalescing	Ovlerap ratio 0.01	Ovlerap ratio 0.2	Ovlerap ratio 0.4	Ovlerap ratio 0.8
0.27673	0.31638	0.305373	0.27673	0.160496

Table 7: Effect on F1 scores of coalescing overlapping rectangles that are classified as the same object.

We note that when run with the videos, our classifier picks a large number of overlapping patch rectangles as positives. This causes our F1 scores to be dramatically worse than it should be because false positives are accentuated. We employ an algorithm of coalescing rectangles if they overlap area is more than a certain ratio. Table 7 shows the effect of the overlap ratio on the F1 scores. We coalesce the rectangles if the amount of overlap exceeds the overlap ratio. All the results were obtained using easy.avi.

4.5 Ablative studies on various motion tracking parameters

5 Note on efficiencies and speedups

We used the following to obtain efficiencies during our development process, as well as to obtain efficiencies during the final classification:

- We used integers as labels, instead of strings to save time on map indexing using the labels in many places in the code.
- We used our 'tune' utility to whet our features on the static images using 4-fold cross validation before using it on the movies.
- We used Lucas Kanade based interpolation techniques to predict the rectangles in skipped frames to speeden up the testing on movies.

References

[1] Bradski G. and Adrian Kaehler. Learning opencv, 2008.

[2] Andrew Ng. Boosting lecture notes cs221, 2009.

6 Appendix

6.1 Tune utility details

6.1.1 Command options

A number of command line options are provided to the tune utility. Some are documented here:

- -examples <integer>: number of examples to use
- -fold <integer> : number of folds to use
- -onefold : boolean that specifies only to do cross-validation on the first of 'k' folds (for quickly testing out new features)
- -depth <integer>: max depth of decision tree to use
- -trees <integer>: number of trees to use for boosting
- -homegrownboost: specifies which version of the classifier is to be used
- -trainerror: option to spit out training error
- -circle_feature, -corner_feature, -edge_feature, -sobel_feature etc: include the various features

6.1.2 Sample output of the tool

Together, these provide us with a valuable tool to diagnose problems before running our classifier on the more time consuming movies. An example of the output of the code is given below:

```
$ make tune

$ ./tune -homegrownboost -trainerror -fold 4 -examples 16000 -depth 1
-trees 400 -files /afs/ir/class/cs221/vision/data/vision_all
=====================================================
Using *HOMEGROWN* boosting classifier
Using 400 trees in boosting
Using 1 depth in boosting
Using 4 folds for validation.
Using files in /afs/ir/class/cs221/vision/data/vision_all
Using 16000 examples
Running experiments on total of: 16000 files
=====================================================
Average test errors:
Fold 0: Test error: 0.0185 Training error: 0
Time taken in fold : 166.64 seconds
Fold 1: Test error: 0.0155 Training error: 0
Time taken in fold : 176.69 seconds
Fold 2: Test error: 0.02125 Training error: 0
Time taken in fold : 177.07 seconds
Fold 3: Test error: 0.0175 Training error: 0
Time taken in fold : 175.28 seconds
——————————————————
Avg Training error Avg Test error
0 0.0181875
———————————————————————-
Confusion Matrix: Predicted labels ->
———————————————————————-
mug stapl keybo clock sciss other
———————————————————————-
mug 60 0 2 0 0 32
stapl 3 85 3 0 1 42
keybo 1 0 63 1 2 25
clock 0 0 0 11 0 4
sciss 0 0 3 0 77 25
other 27 32 27 13 48 15413
———————————————————————-
Prec Recall F-1
———————————————————————-
mug 0.659 0.638 0.649
stapl 0.726 0.634 0.677
keybo 0.643 0.685 0.663
clock 0.440 0.733 0.550
sciss 0.602 0.733 0.661
other 0.992 0.991 0.991
Time taken in entire experiment: 695.68 seconds
```