

Pattern Recognition: Coursework 1

Malhar Jajoo
Imperial College London
mj2514@ic.ac.uk

Armando Piasko
Imperial College London
armando.piasko17@ic.ac.uk

Abstract

The report contains MATLAB implementation of Principal Component Analysis (PCA) and Support Vector Machine (SVM). It explores various properties of PCA for facial image reconstruction as well as classification.

SVM is extended for Multi-Class classification of facial images. Finally, classification results of PCA and SVM are compared.

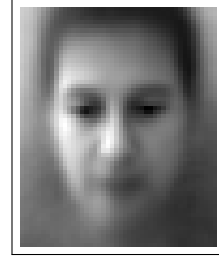


Figure 1: Mean image for input data.

1. Principal Component Analysis

1.1. Introduction

Principal Component Analysis (PCA) is a successful dimensionality reduction algorithm, whose central output is a set of basis vectors in a reduced dimension. These basis vectors are the eigenvectors corresponding to k^1 largest eigenvalues of the Covariance matrix of input Data.

It is a Generative Algorithm² and hence can be used for reconstruction. This property is further used for classification.

1.2. Eigenfaces

In the following section, two methods (varying in computation complexity) are demonstrated for applying PCA to input face data.

The input face data is partitioned into a training and test set in the ratio of 4:1 respectively. This ratio has been proven to be a good rule of thumb. The partitioning also ensures that training set contains some examples of each class.

¹If input feature x is of dimension n , the output of PCA is a reduced subspace of dimension k such that (ideally) $k \ll n$.

²A Generative Algorithm learns joint probability distribution for input data x and labels y as $p(x, y)$ whereas a Discriminative Algorithm learns conditional probability distribution $p(y|x)$.

1.2.1 PCA: computationally intensive method

In this section, the Covariance matrix S is calculated as follows.

$$S_{2576 \times 2576} = \frac{1}{N} (X_{2576 \times 416} \times X^T) \quad (1)$$

where X is mean-centered³ input data matrix and N is total number of observations.

The mean image⁴ is shown in figure 1

Decomposing Covariance matrix

The Covariance matrix is calculated as shown above, and then Matlab SVD command is used to obtain eigenvectors and eigenvalues of the matrix.

The first ten eigenvectors are shown in figure 2. Each eigenvector accounts for a set of features that characterize variations among input faces.

Eigenvalues after $N-1$, where N is the number of observations (416 for partition used), are very close to zero and hence these eigenvalues and corresponding eigenvectors are discarded.

³mean along each dimension of the input data matrix is calculated and subtracted from input matrix for all observations.

⁴This image is obtained by reshaping mean vector into a matrix of size 46×56 . Henceforth all images will be of this dimension unless stated otherwise.

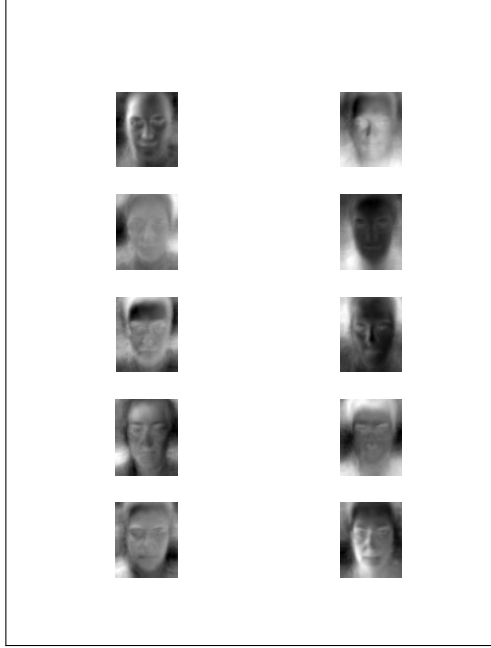


Figure 2: First ten eigenvectors of covariance matrix. They look like ghostly faces and are also called eigenfaces.

Heuristic for choosing PCA dimension k

After obtaining non-zero eigenvalues and corresponding eigenvectors through the process shown above, a heuristic is required to choose the dimension of the resultant subspace, which should be lesser than that of the input data.⁵

The value of k is determined by computing a ratio for increasing values of k until it falls below a (user-calibrated) threshold. The smallest value of k thus obtained is chosen.

More details on the algorithm is provided as Appendix code listing 1.

The resultant value found using the heuristic is

$$k = 124$$

The output of PCA is a set of basis vectors W given by -

$$W_{2576 \times k} = \text{first } k \text{ eigenvectors of } S$$

1.2.2 PCA: Efficient method

In this section, the Covariance matrix S is calculated as follows.

$$S_{416 \times 416} = \frac{1}{N}(X^T \times X_{2576 \times 416}) \quad (2)$$

⁵The input data after partitioning, as explained in section 1.2 is of size 2576×416 in this coursework. PCA aims to determine a value $k \ll 416$.

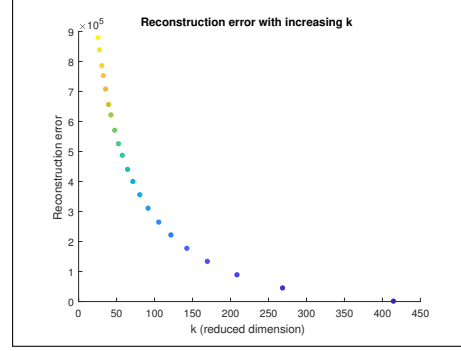


Figure 3: Reconstruction error is inversely proportional to k .

Comparing dimension of S in equation (2) with S in equation (1), it is clear that this method is much faster than the previous method.

The eigenvalues and eigenvectors are computed as explained in section 1.2.1. It is found that the eigenvalues thus computed are the same as first $N - 1$ eigenvalues computed in section 1.2.1. The eigenvectors are related by

$$a_{2576 \times 1} = X_{2576 \times 416} \times b_{416 \times 1},$$

where a refers to an eigenvector calculated as in 1.2.1, b refers to an eigenvector calculated as in 1.2.2 and X is training data.

It is important to normalize the eigenvectors a obtained in this method since they need to be basis vectors with unit magnitude.

1.3. Application of Eigenfaces

In the following section, input face images are reconstructed using results (eigenvectors) from PCA. This property is further used for classification of test images.

1.3.1 Face Image Reconstruction

The theoretical Reconstruction error J is given by

$$J = \sum_{i=k+1}^N \text{eigenvalues} \quad (3)$$

The practical Reconstruction error is given by

$$P = \frac{1}{N} \sum_{n=1}^N |x_n - x_{n \text{ reconstructed}}|^2 \quad (4)$$

It is found that both (3) and (4) lead to same

$$\text{Reconstruction error} = 216758.39$$

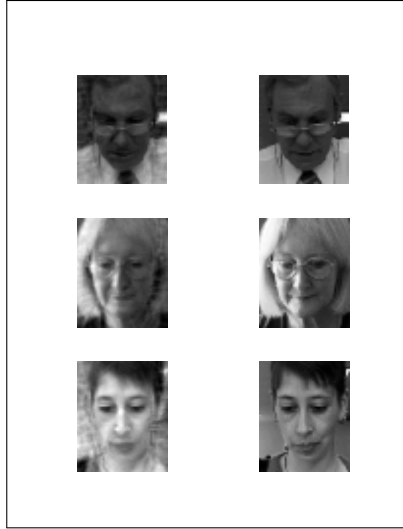


Figure 4: The right column contains face images from training set and left column contains corresponding reconstructed images with $k = 124$.

An example of 3 reconstructed images from training set and test set is given in figure 4 and figure 5 (in the Appendix) respectively.

Reconstructed image is calculated using following formula

$$\text{Reconstructed image} = W * y_i + \text{mean},$$

where y_i is PCA score, W refers to PCA subspace.

1.3.2 PCA-based Classification

In this section, PCA is used for classification of images using procedure from slide 23 of PRlecturefacesubspace (lecture notes). In brief, first the eigensubspaces for all classes is computed, then each test sample is projected on each eigensubspace. Finally, the class with the least Reconstruction error is chosen as prediction.

Metric	Value
Prediction Accuracy	60.46154%
Precision	0.5292
Recall	0.69444
Specificity	0.79297

Table 1: Evaluating classifier performance. Various statistics calculated from confusion matrix. These values are averaged over total classes.

2. Support Vector Machine

2.1. Introduction

Support Vector Machines (SVM) is a highly successful Binary Classification algorithm. It is a supervised learning algorithm that aims at maximizing margin.

2.2. Multi-Class Extension for SVM

In this section, Multi-class extension of SVM is explored using one-vs-all (OvA) and one-vs-one (OvO) methods and then classification results are compared with PCA-based classification.

The classifier is first trained on the *original* training set and then trained on the *reconstructed* training set (obtained using PCA).

The input data partition is the same as used in section 1.2.

Matlab code (containing explanation in comments) for OvA and OvO methods is provided as Appendix code listing 2 and 3 respectively. For Gaussian (RBF) kernel, this code depends on inputs from Cross Validation, which is discussed below.

Selecting Model Parameters

In the section below, Linear and Gaussian Kernel models are explored as candidates for Kernel functions. The Gaussian (RBF) Kernel requires two parameters, "C" and "sigma" which are chosen using a principled approach for Model selection known as k-fold Cross Validation, where k is usually 5 or 10.

k-fold Cross Validation is a technique that improves the generalization (out-of-sample performance) of the model to an unknown/test dataset. This technique reduces the problem of Overfitting⁶ to an extent as it does not rely on the training error for selection of parameters.

Matlab code (containing explanation in comments) for 5-fold Cross Validation is provided as Appendix code listing 4.

2.3. Evaluating Classifier performance

In the following section, Classifier performance is evaluated for Linear and Gaussian Kernel, using various performance metrics, each of which is provided in a tabular form.

Please note following

- Metrics derived from the Confusion Matrix (Accuracy, Precision, Recall and Specificity) are averaged over all classes in the test set. Code for this is provided as Appendix code listing 5.

⁶Overfitting occurs when model parameters capture underlying structure of training data *too well* preventing it from generalizing well to an unknown test set whose structure differs largely from the training set.

- All data has been standardized⁷ before training SVM classifiers.
- For RBF kernel, Cross validation, over the entire training set was employed to find the best value of $C=32$ and $\sigma=128$.

SVM Classification using Training set

Metric	Value	Metric	Value
Training time	3.6124s	Training time	13.546577s
Testing time	0.345s	Testing time	8.659154s
Accuracy	90.384615%	Accuracy	81.730769%
Precision	0.880556	Precision	0.832639
Recall	0.9	Recall	0.849306
Specificity	1.018736	Specificity	1.016893

Table 2: Various metrics calculated for **Gaussian Kernel** Model. The left table contains results on using OvA extension for SVM and right table contains results on using OvO extension for SVM.

Metric	Value	Metric	Value
Training time	2.230037s	Training time	12.920337s
Testing time	0.335734s	Testing time	6.947587s
Accuracy	92.307692%	Accuracy	83.7679 %
Precision	0.910069	Precision	0.853125
Recall	0.931944	Recall	0.852778
Specificity	1.019161	Specificity	0.996061

Table 3: Various metrics calculated for **Linear Kernel** Model. The left table contains results on using OvA extension for SVM and right table contains results on using OvO extension for SVM.

Linearly Separable Data

While experimenting with the classifier, it was found that the input data is linearly separable⁸. This was found by providing the input data as training and test set that resulted in an accuracy of 100%.

As can be seen above, Linear kernel has a better performance (speed and correctness) than a cross-validated Gaussian Kernel.

Hence, for the PCA reconstructed training set below, only results for Linear Kernel are provided.

SVM Classification using PCA Reconstructed set

Metric	Value	Metric	Value
Training time	2.353838s	Training time	12.920337s
Testing time	0.373059s	Testing time	6.947587s
Accuracy	91.346154%	Accuracy	81.730769%
Precision	0.900347	Precision	0.840972
Recall	0.919792	Recall	0.852778
Specificity	1.018962	Specificity	0.996052

Table 4: Usage of PCA reconstructed training set on **Linear Kernel** Model results in slight decrease in performance compared to 3

A few conclusions can be drawn based on the data in the Tables 2,3,4 above.

- one-vs-all method outperformed one-vs-one in both speed and accuracy. A possible reason can be the small size of training set for each pair of classifiers in the one-vs-one method.
- Since data is linearly separable, Linear kernel performs better than Gaussian kernel. Gaussian kernel can result in overfitting of data even after cross validation.
- Usage of PCA reconstructed training set results in slight decrease in performance(Accuracy,precision,recall) compared to linear kernel model. Training and test times remain comparable.

3. Conclusion

Based on all the results in PCA and SVM sections, it can be concluded that PCA, being a Generative Algorithm, is more suited towards Dimensionality Reduction/Reconstruction problems while SVM being a Discriminative algorithm is better suited for Classification problems.

⁷standardization can greatly reduce the time required for training phase.

⁸refers to n -dimensional sample points that can be classified by a $n - 1$ dimensional hyperplane such that there are no outliers.

Appendices

A. PCA

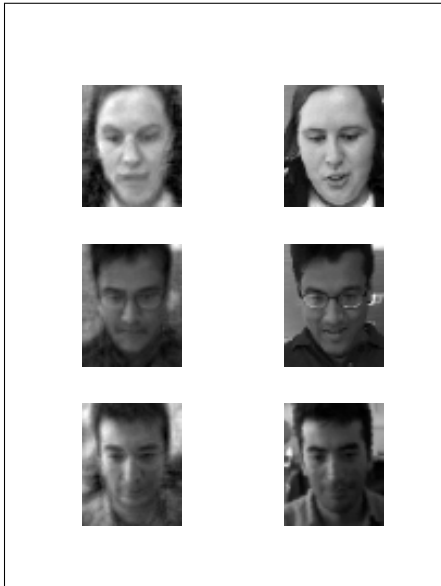


Figure 5: The right column contains face images from test set and left column contains corresponding reconstructed images (using PCA).

Listing 1: Code for heuristic for choosing k .

```

1
2 % Method(Theoretical) - Calculate k based on thresholding a ratio.
3 % inputs - max_dim: max value of k ( this is N-1, where N is number of training
4 %           eigenvalues: eigenvalues of Covariance Matrix.
5
6 % output - best value of k that makes the ratio(recon_error/variance) <=
7 % threshold.
8
9 % This method was based on Machine Learning course by Andrew Ng -
10 % https://www.coursera.org/learn/machine-learning/lecture/S1bq1/choosing-the-
    number-of-principal-components
11 function [best_k,best_reconstruction_error] = choose_bestK(max_dim,eigvalues)
12
13     % 95 % variance retained
14     threshold = 0.05;
15
16     % For each k value, find the ratio as shown below.
17     % As we keep increasing "k", the ratio numerator
18     % decreases and the denominator increases and hence
19     % overall ratio decreases and converges to "threshold" before
20     % the loop iterations reach max_dim .
21     best_k = -1;
22     best_reconstruction_error = -1;
23
24     variance = sum(eigvalues);
25
26     % imp - k values need to be in ascending order
27     % since we want to find the smallest "k" that
28     % is lesser than the "threshold" defined above.
29
30     for k = 1:max_dim % [1,2,3,4...415]
31         reconstruction_error = sum(eigvalues(k+1:end));
32         ratio = reconstruction_error/variance;
33
34         if ratio <= threshold
35             % storing error and variance for value of chosen "k"
36             best_k = k;
37             best_reconstruction_error = reconstruction_error;
38
39             break;
40         end
41     end
42 end

```

B. SVM

Listing 2: Code for one-vs-all(OvA) extension for multi-class SVM.

```
1 function [accuracy] = one_vs_all(train_set,train_labels,test_set,test_labels,bestC
   ,bestSigma)
2 % Summary
3 % This function extends SVM to multi-class SVM by learning 52 classifiers,
4 % one for each class (Based on partitioning, training data always has at
5 % least 1 sample belonging to each class ).
6
7 % The parameters "C" and "sigma" used during learning are obtained from
8 % 5-fold cross validation. See kFoldCrossValidation.m for details on procedure.
9
10 % After training, for each test sample, the classifier with the maximum
11 % "score" is found, and the corresponding class label is chosen as the
12 % predictin label. Here "score" refers to an continuous value, not a
13 % binary label of {-1,+1}.
14
15
16 % Imp: maintain mapping between indices and actual classes.
17 % eg: if there are 30 unique classes in training labels, but their
18 % value is from , eg: 70 to 100. This is not the case in the input
19 % data for this cw but for other cases, will be needed.
20 idx2Class = unique(train_labels);
21 total_classes = size(idx2Class,1);
22
23 tic; % start timer for training phase.
24
25 % ===== One-vs-all ( Training Phase ) =====
26
27 % SVMModels stores all 52 learnt classifiers.
28 SVMModels = cell(total_classes,1);
29
30 % training "total_classes" classifiers
31 for index = 1:total_classes
32
33     % imp, cannot assume input set has all classes in a given range.
34     % Hence safer to use an index to find out class.
35     currentClass = idx2Class(index);
36
37     % for each class, convert all y labels of current class
38     % to +1, and remaining labels to -1
39
40     Y = train_labels;
41     Y(train_labels == currentClass)= 1;
42     Y(train_labels ~= currentClass)= -1;
43
44     % Note - Imp to standardize the training data set.
45     SVMModels{index} = fitcsvm(train_set,Y,'Standardize',true,...
46         'KernelFunction','linear');
47
48     %SVMModels{index} = fitcsvm(train_set,Y,'Standardize',true,...
49     % 'KernelFunction','rbf','KernelScale',bestSigma,'BoxConstraint',bestC);
```

```

50
51     end
52
53     toc;
54
55
56     tic;      % start timer for testing phase.
57     % ===== One-vs-all ( Testing Phase ) =====
58
59     % Matrix storing scores (for positive class) by all classifiers for
60     % each test sample.
61     Scores = zeros(size(test_set,1),total_classes);
62
63     % For each trained SVM, get the score for the "positive" class.
64     for index = 1:total_classes
65         [~,score] = predict(SVMModels{index},test_set);
66         % Second column contains positive-class scores
67         Scores(:,index) = score(:,2);
68     end
69
70     % Most imp, use the max score( max of all classifiers )
71     % to obtain indices of the predicted classes.
72     [~,maxScoreIndices] = max(Scores,[],2);
73
74     predictorLabels = idx2Class(maxScoreIndices);
75     toc;
76
77
78     % ===== Performance metric summary =====
79
80     [accuracy,precision,recall,specificity] = EvaluateClassifier(predictorLabels,
81         test_labels);
82     fprintf(' (accuracy,precision,recall,specificity) of OVA = (%f %%,%f,%f,%f)\n',
83         accuracy,precision,recall,specificity);
84
85 end
86
87
88
89 % Compares prediction labels with given labels for a test dataset
90 % and calculates accuracy as a percentage value.
91 function [acc] = findAccuracy(predictorLabels, testLabels)
92     N = size(testLabels,1);
93     acc = (sum(predictorLabels == testLabels )/N)*100;
94 end

```


Listing 3: Code for one-vs-one(OvO) extension for multi-class SVM.

```

1 function [accuracy] = one_vs_one(train_samples,train_labels,test_samples,
2     test_labels,bestC,bestSigma)
3
4     % imp to maintain mapping between indices and actual classes.
5     % eg: if there are 30 unique classes in training labels, but their
6     % value is from , eg: 70 to 100.
7     idx2Class = unique(train_labels);
8     total_classes = size(idx2Class,1); % <=52(since depends on partition) in our
9         case.
10
11     tic; % start timer for training phase.
12
13     % ===== One-vs-one ( Training Phase ) =====
14     % 1) For each pair of classes, first extract training labels
15     % and corresponding samples.
16
17     % 2) Then convert all y labels of first class
18     % to +1, and labels of other class to -1
19
20     SVMModel_cell = cell(1326,1);
21
22     % Iterate over all possible combinations
23     % ( in an efficient manner, looks similar to bubble sort )
24     % and train 52C2 = 1326 SVM classifiers.
25     idx1 = 1;
26     for i = 1:total_classes
27         for j = i+1:total_classes
28
29             % unlike OvA method, we need to filter the training labels
30             % and samples based on class pair in current iteration.
31             indices = train_labels == i | train_labels == j ;
32             training_samples12 = train_samples(indices,:);
33             temp_labels12 = train_labels(indices);
34
35             % convert class labels to +1 or -1 depending on positive class.
36             % Here the "i" class is the positive class.( and "j" is negative)
37             training_labels12 = temp_labels12;
38             training_labels12(temp_labels12 == i)= 1;
39             training_labels12(temp_labels12 == j)= -1;
40
41             % uncomment 1 of the 2 lines below to use either linear or gaussian
42             % kernel.
43             SVMModel_cell{idx1} = fitcsvm(training_samples12,training_labels12, '
44                 Standardize',true,...
45                 'KernelFunction','linear');
46             %SVMModel_cell{idx1} = fitcsvm(training_samples12,training_labels12, '
47                 Standardize',true,'KernelFunction','rbf',...
48                 'KernelScale',bestSigma,'BoxConstraint',bestC);
49
50             idx1 = idx1 + 1;
51         end
52     end
53
54     toc; % stop timer for training phase.
55
56     % ===== One-vs-one ( Testing Phase ) =====
57     % 1) For each pair of classes, first extract test labels
58     % and corresponding samples.
59
60     % 2) Then convert all y labels of first class
61     % to +1, and labels of other class to -1
62
63     test_idx1 = 1;
64     for i = 1:total_classes
65         for j = i+1:total_classes
66
67             % unlike OvA method, we need to filter the test labels
68             % and samples based on class pair in current iteration.
69             test_indices = test_labels == i | test_labels == j ;
70             test_training_samples12 = test_samples(test_indices,:);
71             test_temp_labels12 = test_labels(test_indices);
72
73             % convert class labels to +1 or -1 depending on positive class.
74             % Here the "i" class is the positive class.( and "j" is negative)
75             test_training_labels12 = test_temp_labels12;
76             test_training_labels12(test_temp_labels12 == i)= 1;
77             test_training_labels12(test_temp_labels12 == j)= -1;
78
79             % uncomment 1 of the 2 lines below to use either linear or gaussian
80             % kernel.
81             test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
82                 Standardize',true,...
83                 'KernelFunction','linear');
84             %test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
85                 Standardize',true,'KernelFunction','rbf',...
86                 'KernelScale',bestSigma,'BoxConstraint',bestC);
87
88             test_idx1 = test_idx1 + 1;
89         end
90     end
91
92     % ===== One-vs-one ( Accuracy ) =====
93     % 1) For each pair of classes, first extract test labels
94     % and corresponding samples.
95
96     % 2) Then convert all y labels of first class
97     % to +1, and labels of other class to -1
98
99     test_idx1 = 1;
100    for i = 1:total_classes
101        for j = i+1:total_classes
102
103            % unlike OvA method, we need to filter the test labels
104            % and samples based on class pair in current iteration.
105            test_indices = test_labels == i | test_labels == j ;
106            test_training_samples12 = test_samples(test_indices,:);
107            test_temp_labels12 = test_labels(test_indices);
108
109            % convert class labels to +1 or -1 depending on positive class.
110            % Here the "i" class is the positive class.( and "j" is negative)
111            test_training_labels12 = test_temp_labels12;
112            test_training_labels12(test_temp_labels12 == i)= 1;
113            test_training_labels12(test_temp_labels12 == j)= -1;
114
115            % uncomment 1 of the 2 lines below to use either linear or gaussian
116            % kernel.
117            test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
118                Standardize',true,...
119                'KernelFunction','linear');
120            %test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
121                Standardize',true,'KernelFunction','rbf',...
122                'KernelScale',bestSigma,'BoxConstraint',bestC);
123
124            test_idx1 = test_idx1 + 1;
125        end
126    end
127
128    % ===== One-vs-one ( Accuracy ) =====
129    % 1) For each pair of classes, first extract test labels
130    % and corresponding samples.
131
132    % 2) Then convert all y labels of first class
133    % to +1, and labels of other class to -1
134
135    test_idx1 = 1;
136    for i = 1:total_classes
137        for j = i+1:total_classes
138
139            % unlike OvA method, we need to filter the test labels
140            % and samples based on class pair in current iteration.
141            test_indices = test_labels == i | test_labels == j ;
142            test_training_samples12 = test_samples(test_indices,:);
143            test_temp_labels12 = test_labels(test_indices);
144
145            % convert class labels to +1 or -1 depending on positive class.
146            % Here the "i" class is the positive class.( and "j" is negative)
147            test_training_labels12 = test_temp_labels12;
148            test_training_labels12(test_temp_labels12 == i)= 1;
149            test_training_labels12(test_temp_labels12 == j)= -1;
150
151            % uncomment 1 of the 2 lines below to use either linear or gaussian
152            % kernel.
153            test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
154                Standardize',true,...
155                'KernelFunction','linear');
156            %test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
157                Standardize',true,'KernelFunction','rbf',...
158                'KernelScale',bestSigma,'BoxConstraint',bestC);
159
160            test_idx1 = test_idx1 + 1;
161        end
162    end
163
164    % ===== One-vs-one ( Accuracy ) =====
165    % 1) For each pair of classes, first extract test labels
166    % and corresponding samples.
167
168    % 2) Then convert all y labels of first class
169    % to +1, and labels of other class to -1
170
171    test_idx1 = 1;
172    for i = 1:total_classes
173        for j = i+1:total_classes
174
175            % unlike OvA method, we need to filter the test labels
176            % and samples based on class pair in current iteration.
177            test_indices = test_labels == i | test_labels == j ;
178            test_training_samples12 = test_samples(test_indices,:);
179            test_temp_labels12 = test_labels(test_indices);
180
181            % convert class labels to +1 or -1 depending on positive class.
182            % Here the "i" class is the positive class.( and "j" is negative)
183            test_training_labels12 = test_temp_labels12;
184            test_training_labels12(test_temp_labels12 == i)= 1;
185            test_training_labels12(test_temp_labels12 == j)= -1;
186
187            % uncomment 1 of the 2 lines below to use either linear or gaussian
188            % kernel.
189            test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
190                Standardize',true,...
191                'KernelFunction','linear');
192            %test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
193                Standardize',true,'KernelFunction','rbf',...
194                'KernelScale',bestSigma,'BoxConstraint',bestC);
195
196            test_idx1 = test_idx1 + 1;
197        end
198    end
199
200    % ===== One-vs-one ( Accuracy ) =====
201    % 1) For each pair of classes, first extract test labels
202    % and corresponding samples.
203
204    % 2) Then convert all y labels of first class
205    % to +1, and labels of other class to -1
206
207    test_idx1 = 1;
208    for i = 1:total_classes
209        for j = i+1:total_classes
210
211            % unlike OvA method, we need to filter the test labels
212            % and samples based on class pair in current iteration.
213            test_indices = test_labels == i | test_labels == j ;
214            test_training_samples12 = test_samples(test_indices,:);
215            test_temp_labels12 = test_labels(test_indices);
216
217            % convert class labels to +1 or -1 depending on positive class.
218            % Here the "i" class is the positive class.( and "j" is negative)
219            test_training_labels12 = test_temp_labels12;
220            test_training_labels12(test_temp_labels12 == i)= 1;
221            test_training_labels12(test_temp_labels12 == j)= -1;
222
223            % uncomment 1 of the 2 lines below to use either linear or gaussian
224            % kernel.
225            test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
226                Standardize',true,...
227                'KernelFunction','linear');
228            %test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
229                Standardize',true,'KernelFunction','rbf',...
230                'KernelScale',bestSigma,'BoxConstraint',bestC);
231
232            test_idx1 = test_idx1 + 1;
233        end
234    end
235
236    % ===== One-vs-one ( Accuracy ) =====
237    % 1) For each pair of classes, first extract test labels
238    % and corresponding samples.
239
240    % 2) Then convert all y labels of first class
241    % to +1, and labels of other class to -1
242
243    test_idx1 = 1;
244    for i = 1:total_classes
245        for j = i+1:total_classes
246
247            % unlike OvA method, we need to filter the test labels
248            % and samples based on class pair in current iteration.
249            test_indices = test_labels == i | test_labels == j ;
250            test_training_samples12 = test_samples(test_indices,:);
251            test_temp_labels12 = test_labels(test_indices);
252
253            % convert class labels to +1 or -1 depending on positive class.
254            % Here the "i" class is the positive class.( and "j" is negative)
255            test_training_labels12 = test_temp_labels12;
256            test_training_labels12(test_temp_labels12 == i)= 1;
257            test_training_labels12(test_temp_labels12 == j)= -1;
258
259            % uncomment 1 of the 2 lines below to use either linear or gaussian
260            % kernel.
261            test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
262                Standardize',true,...
263                'KernelFunction','linear');
264            %test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
265                Standardize',true,'KernelFunction','rbf',...
266                'KernelScale',bestSigma,'BoxConstraint',bestC);
267
268            test_idx1 = test_idx1 + 1;
269        end
270    end
271
272    % ===== One-vs-one ( Accuracy ) =====
273    % 1) For each pair of classes, first extract test labels
274    % and corresponding samples.
275
276    % 2) Then convert all y labels of first class
277    % to +1, and labels of other class to -1
278
279    test_idx1 = 1;
280    for i = 1:total_classes
281        for j = i+1:total_classes
282
283            % unlike OvA method, we need to filter the test labels
284            % and samples based on class pair in current iteration.
285            test_indices = test_labels == i | test_labels == j ;
286            test_training_samples12 = test_samples(test_indices,:);
287            test_temp_labels12 = test_labels(test_indices);
288
289            % convert class labels to +1 or -1 depending on positive class.
290            % Here the "i" class is the positive class.( and "j" is negative)
291            test_training_labels12 = test_temp_labels12;
292            test_training_labels12(test_temp_labels12 == i)= 1;
293            test_training_labels12(test_temp_labels12 == j)= -1;
294
295            % uncomment 1 of the 2 lines below to use either linear or gaussian
296            % kernel.
297            test_SVMModel_cell{test_idx1} = fitcsvm(test_training_samples12,test_training_labels12, '
298                Standardize',true,...
29
```

```

48
49     end
50 end
51
52 toc;
53
54
55
56 tic; % start timer for testing phase.
57
58 % ===== One-vs-one ( Testing Phase )=====
59 % A "voting/histogram" method for predictions -
60
61 % This is done by using a hashtable with a row for each
62 % class and a column for each test sample, for storing "votes".
63
64 % Each vote is generated based on prediction by a single classifier ( see
    below
65 % inside the two nested for-loops ). The corresponding entry of the hashtable
66 % is then incremented.
67
68 % Finally, predictions are extracted from the hash table
69 HashTable = zeros(total_classes,size(test_samples,1));
70 idx2 = 1;
71
72 % loop over all classifiers.
73 for i = 1:total_classes
74     for j = i+1:total_classes
75
76         % unlike one-vs-all, we don't need the prediction score here.
77         [labels,~] = predict(SVMModel_cell{idx2},test_samples);
78
79         % For each prediction, increment entries of hashtable by 1
80         for k = 1:size(labels,1)
81             if(labels(k)==1)
82                 HashTable(i,k) = HashTable(i,k) + 1;
83             else
84                 HashTable(j,k) = HashTable(j,k) + 1;
85             end
86         end
87
88         idx2 = idx2 +1;
89     end
90 end
91
92 % Convert Hashtable to prediction labels
93 predLabels = zeros(size(test_samples,1),1);
94
95 % Gather labels from each column of the hashtable
96 for i = 1:size(HashTable,2)
97     [~,idx] = max(HashTable(:,i));
98     predLabels(i,1) = idx2Class(idx);
99 end

```

```

100
101
102
103     toc;
104
105
106
107     % ===== Performance metric summary =====
108
109     [accuracy,precision,recall,specificity] = EvaluateClassifier(predLabels,
110         test_labels);
111     fprintf(' (accuracy,precision,recall,specificity) of OVO = (%f %% ,%f,%f,%f)\n'
112         ,accuracy,precision,recall,specificity);
113
114 end
115
116 % ===== Helper function =====
117 % Compares prediction labels with given labels for a test dataset
118 % and calculates accuracy as a percentage value.
119 function [acc] = findAccuracy(predictorLabels, testLabels)
120     N = size(testLabels,1);
121     acc = (sum(predictorLabels == testLabels )/N)*100;
122
123 end

```

Listing 4: Code for k-fold Cross validation.

```

1 function [bestC,bestSigma] = kFoldCrossValidation(train_set,train_labels,numFolds)
2 % Summary
3 % Use K-fold cross validation to find best "C" and "sigma" for a given training
   set
4
5 % Procedure:
6 % Assume numFolds = 5, hence 5-fold cross validation.
7
8 % 1) Partition trainig_set = training_set + validation set. This is done using
9 % matlabs crossvalind() as shown below.
10
11 % 2) Choose a range of C and sigma as shown below.
12
13 % 3) For each pair of (C,sigma), choose a validation set.
14 %     The validation set is chosen "k" times.
15
16 % 4) For each validation set, calculate the binary misclassification error.
17 % Note - Either of one-vs-all or one-vs-one can be used to find error.
18 % Sum it up "k" times and then average it by dividing by "k".
19 % This is the Cross Validation(CV) error.
20
21 % 5) Find the parameters that give the least CV error and this is the final result
   .
22
23 idx2Class = unique(train_labels);
24 total_classes = size(idx2Class,1);
25
26 % separate data into training and validation.
27 % assigns an index between 1-numFolds to each training sample
28 indices = crossvalind('Kfold',train_labels, numFolds);
29
30 % Range of values to search over.
31 C_range = 2.^(-1:1:8);
32 sigma_range = 2.^(1:1:7);
33
34 % Default values
35 bestCV_error = Inf;
36 bestC = -Inf;
37 bestSigma = -Inf;
38
39 % Find combination of (C,sigma) that gives minimum Cross validation error.
40 for C = C_range
41     for sigma = sigma_range
42
43         fprintf('\n (C,sigma) = (%d,%d) %% \n', C,sigma);
44         fprintf(' (bestC,bestSigma) = (%d,%d) %% \n\n', bestC,bestSigma);
45         % Calculate cross validation error for each pair
46         % of given (c,sigma)
47         sum = 0;
48         for i = 1:numFolds
49
50             validation_samples_cv = train_set(indices==i,:);

```

```

51     validation_labels_cv = train_labels(indices==i,:);
52
53     train_samples_cv = train_set(indices~=i,:);
54     train_labels_cv = train_labels(indices~=i,:);
55
56     % find Cross validation (CV) error
57     % Use one-vs-all classification to find CV error.
58     % Note - Either of one-vs-all or one-vs-one can be used to find error.
59
60     % Also, passing C and sigma doesnt imply that the kernel used
61     % is always RBF. These parameters can be (easily) ignored inside the
62     % code of
63     % one_vs_all.m if a linear kernel is being used. Please see the code in
64     % one_vs_all.m
65     acc = one_vs_all(train_samples_cv, train_labels_cv, ...
66                     validation_samples_cv, validation_labels_cv,C,sigma);
67
68     sum = sum + (1 - acc); % since acc is a value between [0,1)
69
70 end
71
72 currentCV_error = sum/numFolds ;
73
74 if( currentCV_error < bestCV_error )
75     bestCV_error = currentCV_error;
76     bestC = C;
77     bestSigma = sigma;
78 end
79 end
end

```

Listing 5: Code for extracting data from Confusion Matrix.

```

1 % Inputs are prediction and ground truth labels
2 % Output are various statistics ( as an average over all classes)
3 % obtained from the confusion matrix.
4 function [accuracy,precision,recall,specificity] = EvaluateClassifier(
    prediction_labels, test_labels)
5
6     total_classes = size(unique(test_labels),1); % 48 for current partition.
7
8     % find accuracy
9     accuracy = findAccuracy(prediction_labels, test_labels);
10    confMatrix = confusionmat(test_labels',prediction_labels');
11
12    sumConfMatrix = sum(confMatrix(:));
13
14    % find precision , take average over all classes
15    precisionSum = 0;
16    recallSum = 0 ;
17    specificitySum = 0 ;
18
19    for i=1:size(confMatrix,1)
20
21        TP = confMatrix(i,i);
22        colSum = sum(confMatrix(:,i));
23        rowSum = sum(confMatrix(i,:));
24
25        % Note - These guards are required since prediction
26        % may not contain a particular class (but ground truth may)
27        if(colSum ~= 0)
28            % precision = TP/(TP+FP)
29            precisionSum = precisionSum + (TP/colSum);
30        end
31
32        % Note - These guards are required since ground truth (test_labels)
33        % may not contain a particular class.(but prediction may)
34        if(rowSum ~= 0)
35            % recall/sensitivity = TP/(TP+FN)
36            recallSum = recallSum + (TP/rowSum);
37        end
38
39        % specificity =
40        TN = sumConfMatrix - rowSum - colSum;
41        val = (TN + (colSum - TP));
42
43        if(val~=0)
44            specificitySum = specificitySum + (TN/val);
45        end
46
47    end
48
49    precision = precisionSum/total_classes;
50    recall = recallSum/total_classes;
51

```

```
52     specificity = specificitySum/total_classes;  
53  
54 end
```

B.1. References

References