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.

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 Algorithm[2] 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.

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

[2]A 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 $p(y|x)$.

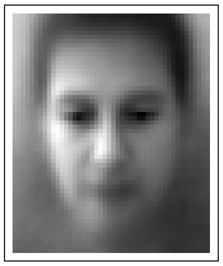

Figure 1: Mean image for input data.

1.2.1 PCA: computationally intensive method

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

$$S_{2576\times2576} = \frac{1}{N}(X_{2576\times416} \times X^T) \qquad (1)$$

where X is mean-centered[3] input data matrix and N is total number of observations.

The mean image[4] 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.

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

[4]This 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.

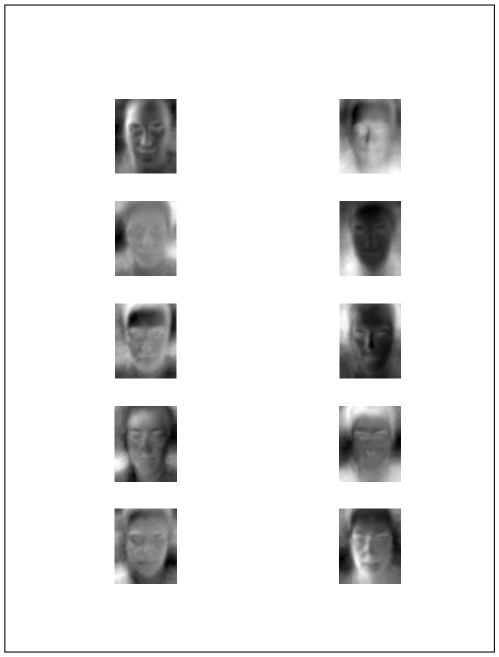

Figure 2: First ten eigenvectors of covariance matrix. They look like ghostly faces and are also called eigenfaces.

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 -

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

1.2.2 PCA: Efficient method

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

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

[5]The 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 <<$416.

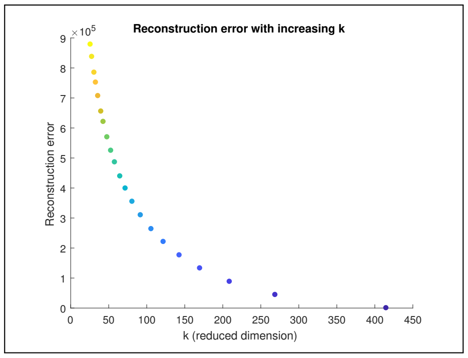

Figure 3: Reconstruction error is inversely proportional to k.

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

$$J = \sum_{i=k+1}^{N} eigenvalues \tag{3}$$

The practical Reconstruction error is given by

$$P = \frac{1}{N}\sum_{n=1}^{N} |x_n - x_{n_{Reconstructed}}|^2 \tag{4}$$

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

$$Reconstructionerror = 216758.39$$

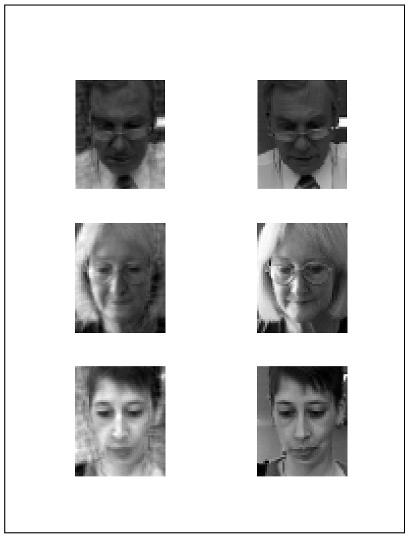

Figure 4: The right column contains face images from training set and left column contains corresponding reconstructed images with k = 124.

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

$$Reconstructedimage = W * y_i + mean,$$

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 Overfitting[6] 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.

[6]Overfitting 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 standardized[7] 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 σ=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 separable[8]. 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.

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

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

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.

Appendices

A. PCA

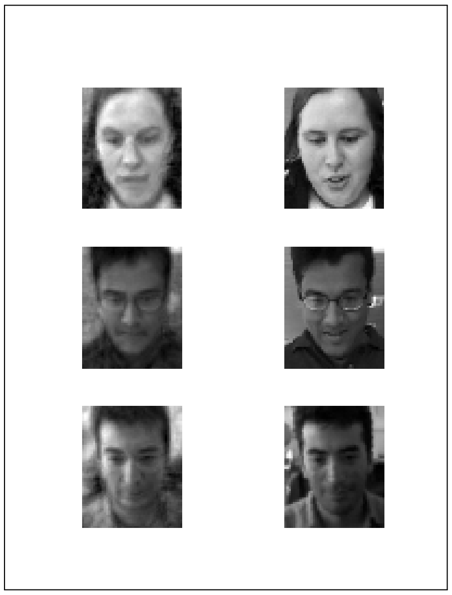

Figure 5: The right column contains face images from test set and left column contains corresponding reconstructed images (using PCA).

Listing 1: Code for heuristic for choosing k.

```

% Method(Theoretical) - Calculate k based on thresholding a ratio.
% inputs - max_dim: max value of k ( this is N-1, where N is number of trainig
    samples).
%           eigenvalues: eigenvalues of Covariance Matrix.

% output - best value of k that makes the ratio(recon_error/variance) <=
% threshold.

% This method was based on Machine Learning course by Andrew Ng -
% https://www.coursera.org/learn/machine-learning/lecture/S1bq1/choosing-the-
    number-of-principal-components
 function [best_k,best_reconstruction_error] = choose_bestK(max_dim,eigvalues)

    % 95 % variance retained
    threshold = 0.05;

    % For each k value, find the ratio as shown below.
    % As we keep increasing "k", the ratio numerator
    % decreases and the denominator increases and hence
    % overall ratio decreases and converges to "threshold" before
    % the loop iterations reach max_dim .
    best_k = -1;
    best_reconstruction_error = -1;

    variance = sum(eigvalues);

    % imp - k values need to be in ascending order
    % since we want to find the smallest "k" that
    % is lesser than the "threshold" defined above.

    for k = 1:max_dim % [1,2,3,4...415]
       reconstruction_error = sum(eigvalues(k+1:end));
       ratio  = reconstruction_error/variance;

        if ratio <= threshold
            % storing error and variance for value of chosen "k"
            best_k = k;
            best_reconstruction_error = reconstruction_error;

            break;
        end
    end
 end
```

B. SVM

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

```
function [accuracy] = one_vs_all(train_set,train_labels,test_set,test_labels,bestC
    ,bestSigma)
% Summary
% This function extends SVM to multi-class SVM by learning 52 classifiers,
% one for each class (Based on partitioning, training data always has at
% least 1 sample belonging to each class ).

% The parameters "C" and "sigma" used during learning are obtained from
% 5-fold cross validation. See kFoldCrossValidation.m for details on procedure.

% After training, for each test sample, the classifier with the maximum
% "score" is found, and the corresponding class label is chosen as the
% predictin label. Here "score" refers to an continuous value, not a
% binary label of {-1,+1}.


    % Imp: maintain mapping between indices and actual classes.
    % eg: if there are 30 unique classes in training labels, but their
    % value is from , eg: 70 to 100. This is not the case in the input
    % data for this cw but for other cases, will be needed.
    idx2Class = unique(train_labels);
    total_classes = size(idx2Class,1);

    tic;    % start timer for training phase.

    % ===== One-vs-all ( Training Phase ) =====

    % SVMModels stores all 52 learnt classifiers.
    SVMModels = cell(total_classes,1);

    % training "total_classes" classifiers
    for index = 1:total_classes

        % imp, cannot assume input set has all classes in a given range.
        % Hence safer to use an index to find out class.
        currentClass = idx2Class(index);

        % for each class, convert all y labels of current class
        % to +1, and remaining labels to -1

        Y = train_labels;
        Y(train_labels == currentClass)=  1;
        Y(train_labels ~= currentClass)= -1;

        % Note - Imp to standardize the training data set.
        SVMModels{index} = fitcsvm(train_set,Y,'Standardize',true,...
            'KernelFunction','linear');

        %SVMModels{index} = fitcsvm(train_set,Y,'Standardize',true,...
        % 'KernelFunction','rbf','KernelScale',bestSigma,'BoxConstraint',bestC);
    end

    toc;


    tic;    % start timer for testing phase.
    % ===== One-vs-all ( Testing Phase ) =====

    % Matrix storing scores (for positive class) by all classifiers for
    % each test sample.
    Scores = zeros(size(test_set,1),total_classes);

    % For each trained SVM, get the score for the "positive" class.
    for index = 1:total_classes
        [~,score] = predict(SVMModels{index},test_set);
        % Second column contains positive-class scores
        Scores(:,index) = score(:,2);
    end

    % Most imp, use the max score( max of all classifiers )
    % to obtain indices of the predicted classes.
    [~,maxScoreIndices] = max(Scores,[],2);

    predictorLabels = idx2Class(maxScoreIndices);
    toc;


    % ===== Performance metric summary ============

    [accuracy,precision,recall,specificity] = EvaluateClassifier(predictorLabels,
       test_labels);
    fprintf('(accuracy,precision,recall,specificity) of OVA = (%f %%,%f,%f,%f)\n',
       accuracy,precision,recall,specificity);


end




% Compares prediction labels with given labels for a test dataset
% and calculates accuracy as a percentage value.
function [acc] = findAccuracy(predictorLabels, testLabels)
    N = size(testLabels,1);
    acc = (sum(predictorLabels ==  testLabels )/N)*100;
end
```

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

```
function [accuracy] = one_vs_one(train_samples,train_labels,test_samples,
   test_labels,bestC,bestSigma)


    % imp to maintain mapping between indices and actual classes.
    % eg: if there are 30 unique classes in training labels, but their
    % value is from , eg: 70 to 100.
    idx2Class = unique(train_labels);
    total_classes = size(idx2Class,1); % <=52(since depends on partition) in our
       case.

    tic; % start timer for training phase.


    % ======= One-vs-one ( Training Phase ) ===========
    % 1) For each pair of classes, first extract training labels
    % and corresponding samples.

    % 2) Then convert all y labels of first class
    % to +1, and labels of other class to -1

    SVMModel_cell = cell(1326,1);

    % Iterate over all possible combinations
    % ( in an efficient manner, looks similar to bubble sort )
    % and train 52C2 = 1326 SVM classifiers.
    idx1 = 1;
    for i = 1:total_classes
        for j = i+1:total_classes

            % unlike OvA method, we need to filter the training labels
            % and samples based on class pair in current iteration.
            indices = train_labels == i | train_labels == j ;
            training_samples12 = train_samples(indices,:) ;
            temp_labels12  = train_labels(indices) ;

            % convert class labels to +1 or -1 depending on positive class.
            % Here the "i" class is the positive class.( and "j" is negative)
            training_labels12 = temp_labels12;
            training_labels12(temp_labels12 == i)= 1;
            training_labels12(temp_labels12 == j)= -1;

            % uncomment 1 of the 2 lines below to use either linear or gaussian
               kernel.
             SVMModel_cell{idx1} = fitcsvm(training_samples12,training_labels12,'
                Standardize',true,...
            'KernelFunction','linear');
             %SVMModel_cell{idx1} = fitcsvm(training_samples12,training_labels12,'
                Standardize',true,'KernelFunction','rbf',...
             %   'KernelScale',bestSigma,'BoxConstraint',bestC);

            idx1 = idx1 + 1;

        end
    end

    toc;



    tic; % start timer for testing phase.

    % ========= One-vs-one ( Testing Phase )===========
    % A "voting/histogram" method for predictions -

    % This is done by using a hashtable with a row for each
    % class and a coloumn for each test sample, for storing "votes".

    % Each vote is generated based on prediction by a single classifier ( see
       below
    % inside the two nested for-loops ). The corresponding entry of the hashtable
    % is then incremented.

    % Finally, predictions are extracted from the hash table
    HashTable = zeros(total_classes,size(test_samples,1));
    idx2 = 1;

    % loop over all classifiers.
    for i = 1:total_classes
        for j = i+1:total_classes

            % unlike one-vs-all, we don't need the prediction score here.
            [labels,~] = predict(SVMModel_cell{idx2},test_samples);

            % For each prediction, increment entries of hashtable by 1
            for k = 1:size(labels,1)
                if(labels(k)==1)
                    HashTable(i,k) = HashTable(i,k) + 1;
                else
                    HashTable(j,k) = HashTable(j,k) + 1;
                end
            end

            idx2 = idx2 +1;
        end
    end

    % Convert Hashtable to prediction labels
    predLabels = zeros(size(test_samples,1),1);

    % Gather labels from each coloumn of the hashtable
    for i = 1:size(HashTable,2)
        [~,idx] = max(HashTable(:,i));
        predLabels(i,1) = idx2Class(idx);
    end
```

```matlab



    toc;



     % ===== Performance metric summary ============

    [accuracy,precision,recall,specificity] = EvaluateClassifier(predLabels,
        test_labels);
    fprintf('(accuracy,precision,recall,specificity) of OVO = (%f %% ,%f,%f,%f)\n'
        ,accuracy,precision,recall,specificity);

end


% ============ Helper function ========================

% Compares prediction labels with given labels for a test dataset
% and calculates accuracy as a percentage value.
function [acc] = findAccuracy(predictorLabels, testLabels)
    N = size(testLabels,1);
    acc = (sum(predictorLabels ==  testLabels )/N)*100;

end
```

Listing 4: Code for k-fold Cross validation.

```
function [bestC,bestSigma] = kFoldCrossValidation(train_set,train_labels,numFolds)
% Summary
% Use K-fold cross validation to find best "C" and "sigma" for a given training
    set

% Procedure:
% Assume numFolds = 5, hence 5-fold cross validation.

% 1) Partitiion trainig_set = training_set + validation set. This is done using
% matlabs crossvalind() as shown below.

% 2) Choose a range of C and sigma as shown below.

% 3) For each pair of (C,sigma), choose a validation set.
%    The validation set is chosen "k" times.

% 4) For each validation set, calculate the binary misclassification error.
% Note - Either of one-vs-all or one-vs-one can be used to find error.
% Sum it up "k" times and then average it by dividing by "k".
% This is the Cross Validation(CV) error.

% 5) Find the parameters that give the least CV error and this is the final result
    .

idx2Class = unique(train_labels);
total_classes = size(idx2Class,1);

% separate data into training and validation.
% assigns an index between 1-numFolds to each training sample
indices = crossvalind('Kfold',train_labels, numFolds);

% Range of values to search over.
C_range = 2.^(-1:1:8);
sigma_range = 2.^(1:1:7);

% Default values
bestCV_error = Inf;
bestC = -Inf;
bestSigma = -Inf;

% Find combination of (C,sigma) that gives minimum Cross validation error.
for C = C_range
   for sigma = sigma_range

        fprintf('\n (C,sigma) = (%d,%d) %% \n', C,sigma);
        fprintf(' (bestC,bestSigma) = (%d,%d) %% \n\n', bestC,bestSigma);
       % Calculate cross validation error for each pair
       % of given (c,sigma)
       sum = 0;
       for i = 1:numFolds

           validation_samples_cv = train_set(indices==i,:);
```

```
            validation_labels_cv = train_labels(indices==i,:);

            train_samples_cv = train_set(indices~=i,:);
            train_labels_cv = train_labels(indices~=i,:);

            % find Cross validation (CV) error
            % Use one-vs-all classification to find CV error.
            % Note - Either of one-vs-all or one-vs-one can be used to find error.

            % Also, passing C and sigma doesnt imply that the kernel used
            % is always RBF. These parameters can be (easily) ignored inside the
               code of
            % one_vs_all.m if a linear kernel is being used. Please see the code in
                one_vs_all.m
            acc = one_vs_all(train_samples_cv, train_labels_cv, ...
                             validation_samples_cv, validation_labels_cv,C,sigma);

            sum = sum + (1 - acc);  % since acc is a value between [0,1)

        end

        currentCV_error = sum/numFolds ;

        if( currentCV_error < bestCV_error )
            bestCV_error = currentCV_error;
            bestC = C;
            bestSigma = sigma;
        end

    end
end
```

Listing 5: Code for extracting data from Confusion Matrix.

```
% Inputs are prediction and ground truth labels
% Output are various statistics ( as an average over all classes)
% obtained from the confusion matrix.
function [accuracy,precision,recall,specificity] = EvaluateClassifier(
   prediction_labels, test_labels)

    total_classes = size(unique(test_labels),1); % 48 for current partition.

    % find accuracy
    accuracy = findAccuracy(prediction_labels, test_labels);
    confMatrix = confusionmat(test_labels',prediction_labels');

    sumConfMatrix = sum(confMatrix(:));

    % find precision , take average over all classes
    precisionSum = 0;
    recallSum = 0 ;
    specificitySum = 0 ;

    for i=1:size(confMatrix,1)

         TP = confMatrix(i,i);
         colSum = sum(confMatrix(:,i));
         rowSum = sum(confMatrix(i,:));

         % Note - These guards are required since prediction
         % may not contain a particular class (but ground truth may)
         if(colSum ~= 0)
             % precision = TP/(TP+FP)
             precisionSum = precisionSum + (TP/colSum);
         end

         % Note - These guards are required since ground truth (test_labels)
         % may not contain a particular class.(but prediction may)
         if(rowSum ~= 0)
             % recall/sensitivity = TP/(TP+FN)
             recallSum = recallSum + (TP/rowSum);
         end

         % specificity =
         TN = sumConfMatrix - rowSum - colSum;
         val = (TN + (colSum - TP));

         if(val~=0)
             specificitySum = specificitySum + (TN/val);
         end


    end

    precision = precisionSum/total_classes;
    recall = recallSum/total_classes;
    specificity = specificitySum/total_classes;

end
```

B.1. References

References