Pattern Recognition Coursework 1

Jakub Mateusz Szypicyn
CID: 00846006
EEE4
jms13@ic.ac.uk

Jacobus Jukka Hertzog
CID: 00828711
EEE4
jjh13@ic.ac.uk

Abstract

In this coursework, we looked at performing face identification using Principal Component Analysis (PCA), the Nearest Neighbour (NN) classifier, and multi-class Support Vector Machines (SVM). We found that we can significantly reduce the execution time of a PCA calculation by using a different method of finding the eigenvectors. Then we investigated the reconstruction error of PCA and looked at how different numbers of eigenvalues affected the performance of the Nearest Neighbour classifier. Finally, we implemented multi-class SVMs using binary SVMs, and compared the performance of different types of SVM.

1. Introduction

It is often desirable to be able to quickly and accurately transform handwritten text into computer text or to assign name to a person based on their face using computer programs. The process requires the computer to have some prior knowledge of what it is trying to compute. This is known as training data. Based on the training data we can build mathematical models which will allows us to recognise faces or letters.

In this paper we are investigating and describing basic methods of training and testing, such as Principal Coefficient Analysis (PCA), Nearest Neighbour classification (NN) and multiclass Support Vector Machine (SVM) classification including binary class SVM.

2. Eigenfaces

2.1. Data partition

A Matlab file containing face data `face.mat` has been provided for the purpose of this coursework. The file contains a 2576×520 matrix of face images. Each image is stored in a column. Given that the matrix has 520 columns there are 520 pictures of faces. Those pictures belong to 52 distinct persons. Therefore there are 10 pictures per person. Furthermore each picture has dimensions of 56×46 pixels.

In order to divide the data set into training and testing subsets, we have decide to preserve as much variance int he training data as possible. This would ensure that each set of faces is separated as far as possible, which potentially ensures higher identification rate.

The data was divided in the following ratio of testing to training: 20% to 80%. From each set of 10 pictures we have thus taken two most average pictures, based on the average pixel values. The two sets will be heron referred to as `training` (2576×416 matrix) and `testing` (2576×104 matrix).

2.2. PCA of face data

2.2.1 AA^T

Following the algorithm for Principal Component Analysis, we have first detrended the face images by subtracting a mean vector from all columns of `training`, which resulted in a matrix A, whose rows are now zero-mean. Following the above, the covariance matrix $S = \frac{1}{416} \times AA^T$ has been calculated. S has dimensions of 2576×2576.

The covariance matrix S uniquely describes the data by calculating its spread or variance denoted σ and its orientation. For face recognition we would like to make use of both of those properties. Namely, we would like to identify and keep vectors along which the data spread is the largest, disposing of dimensions which do not carry any spread information. This helps us to reduce problem size, decrease memory usage and increase performance.

The dimensions corresponding to largest data spread are given to us by calculating the eigenvalues and eigenvectors of S. We expect that there will be at most 416 non-zero eigenvalues. This follows from [1]. Given a rectangular matrix A, $S_1 = AA^T$ and $S_2 = A^TA$ share all non-zero eigenvalues. This means that the larger of the two matrices will have as many non-zero eigenvalues as the smaller one. Given that the dimensions of the smaller matrix are in our case 416×416, we expect that the larger matrix of 2576×2576 will return at most 416 non-zero eigenvalues. It of course can be the case, that there will be fewer non-zero eigenvalues. This proves to be the case with `training`. The resulting covariance matrix produces 415 significant eigenvalues. This can be accredited to one of two things:

1. The data is such that variance in one of the dimensions is actually zero.

2. The precision of `double float` calculations is insufficient. Since the data is very large, none of the 'zero' eigenvalues are actually equal to zero. They are however very small varying between 10^{-10} and 10^{-14}. This is shown in Figure 1 below.

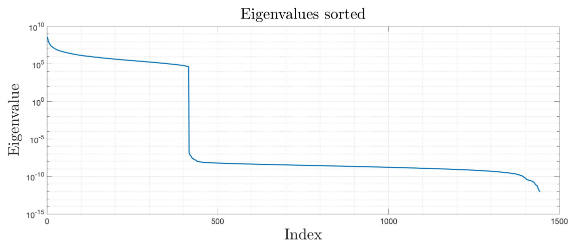

Figure 1. Sorted Eigenvalues of Covariance Matrix S

It can be seen that first 415 values are much greater than 1. The 416th value is around 10^{-10}. The three best eigenvectors, or eigenfaces corresponding to the three highest eigenvalues are shown below in Figure 2 . Finally the mean face which was initially subtracted from the face data is shown in Figure 3 .

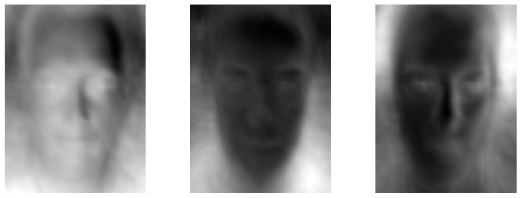

Figure 2. Best 3 Eigenfaces of Covariance Matrix S

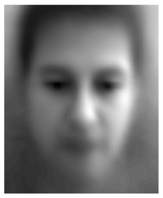

Figure 3. Mean Face from `training`

The 95% confidence bound would require us to use 121 eigenfaces. This constitutes a compromise between accuracy and performance, by reducing the problem dimensionality.

2.2.2 A^TA

Alternatively as suggested earlier we could compute a covariance matrix $S_T = \frac{1}{416} \times A^TA$, which now has dimensions of 416×416 instead of 2576×2576. We know [1] that both matrices produce the same (meaningful) eigenvalues. Their plot in the descending order in Figure 4 proves the above claim.

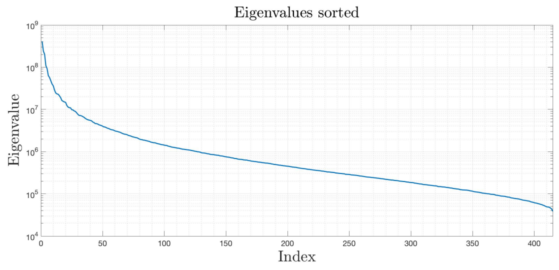

Figure 4. Eignevalues of S_T

Figure 4 allows to to deduce that the 416th value is actually zero. If it was non-zero the two methods of calculating a covariance matrix would result in very tiny, yet identical values. However the first method resulted in eigenvalue 416 being equal to 2.3×10^{-10}, whereas the second method gave a value of -1.9×10^{-12}.

We know however that eigenvectors will be different and therefore more computation is required to find the eigenfaces. Starting from:

$$A^TAx = \lambda x \tag{1}$$

Let us multiply both sides by A:

$$AA^T(Ax) = \lambda(Ax) \tag{2}$$

We deduce that the eigenvectors of AA^T are $u_i = Ax_i, \forall \lambda_i \neq 0$, where x_i is the ith eigenvector of A^TA. Having thus calculated x_i, we must multiply each of the vectors by the original zero-mean matrix `training` in order to obtain eigenfaces.

However as we do it, we find that some of the images have inverted colours - i.e. the direction of the vector has been reversed. using Singular Vector Decomposition for any two matrices AA^T and A^TA we can show that:

$$Av_i = \sigma_i u_i \tag{3}$$

, where u_i and v_i are the eigenvectors of the two matrices respectively and σ_i is the square root of the corresponding eigenvalue. Thus having found v_i we can calculate $u_i = {Av_i}/{\sigma_i}$. Given that $\sigma_i > 0 \forall i$, Av_i has the effect of reversing the direction of some eigenvectors.

Figure 5. Top 3 Eigenfaces

2.2.3 Comparison

We know that the method presented in section 2.2.1 is accurate. It is however more time consuming to calculate the eigenvectors of matrix with dimensions 2576×2576 rather than those of a 416×416 matrix. It is shown [2] that eigenproblem complexity is bounded by $O(n^2log(n) + (nlog^2(n))log(b)$, where b is a measure of accuracy in bits 2^{-b}. However it should be noted that the eigenvectors of the second matrix are not what we are trying to calculate. Thus the latter method introduces an extra step. We have thus timed the full execution of both methods from implementing the `eig` function to having a dataset with properly ordered eigenvectors. The results are shown in Table 1 .

Algorithm	**Time**
AA^T	3.59 sec
A^TA	0.176 sec

Table 1. Computation Time for Both Algorithms

3. Applications of Eigenfaces

3.1. Face reconstruction

For the remainder of the work, the second algorithm (2.2.2) has been used.

Having determined the PCA bases we are now able to reconstruct faces. We expect that those from the training set will be reconstructed more accurately for any given number of PCA components.

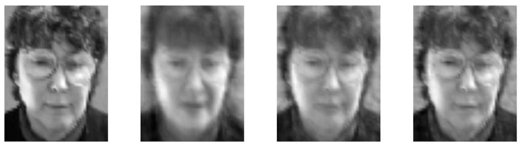

Figure 6. Reconstruction of a Training Face: Original, 50 bases, 121 bases, 180 bases

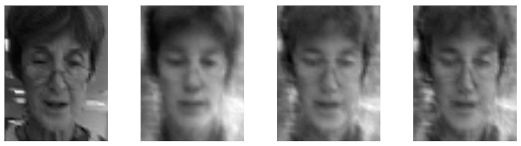

Figure 7. Reconstruction of a Test Face: Original, 50 bases, 121 bases, 180 bases

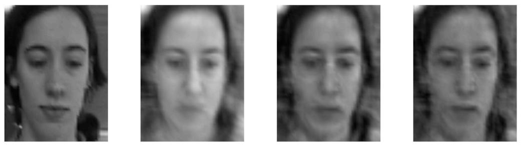

Figure 8. Reconstruction of a Test Face 2: Original, 50 bases, 121 bases, 180 bases

The reconstruction errors measured as the Euclidean norm of the difference of two vectors: $OriginalFace$ and $ReconstructedFace$ of the images shown in Figures 6, 7 and 8 are presented in the Table 2 below.

Image	**Number of bases**	**Reconstruction Error**
Training Face	50	884.4740
Training Face	121	556.3322
Training Face	180	347.7328
Training Face	400	18.1174
Test Face 1	50	864.7057
Test Face 1	121	721.7824
Test Face 1	180	676.7093
Test Face 1	400	582.9752
Test Face 2	50	816.2902
Test Face 2	121	649.3541
Test Face 2	180	575.2193
Test Face 2	400	474.1061

Table 2. Reconstruction error for various images and number of PCA bases

Additionally, we can determine the distortion measure of the reconstruction. For any given number of PCA bases, the distortion measure will be the same for any image from the same training set. The distortion measure has been calculated as: $J = \sum_{n=M+1}^{D} \lambda_n$, where M represents number of PCA bases used to reconstruct the image and D it the total number of eigenfaces. It should also be noted that the eigenvalues are ordered in the descending order s.t. $\lambda_1 > \lambda_2 > ... > \lambda_{M+1} > ... > \lambda_D$.

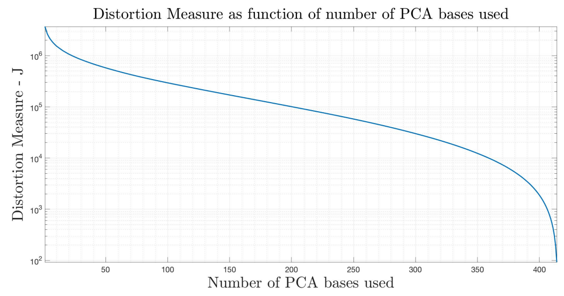

Figure 9. Distrtion Measure for varying number of PCA bases

From the Figures 6, 7 and 8 we can infer that 121 eigenfaces (the 95% bound calculated earlier) is sufficient to a human eye in order to assign class to a given image. 50 PCA bases results in a very fuzzy image, though the main characteristics of the class can be identified. Finally, 180 eigenfaces produce a relatively sharp image, though it isn't much different from the one obtained with 121 bases, as shown in Table 2.

The test results shown in Table 2 show as expected a monotonous trend. As we increase the number of eigenfaces the reconstruction error (or the D-dimensional distance between original and reconstructed images) decreases. We also observe that both face images from the `testing` set produce larger errors than the face from `training` set for larger amount of PCA bases. The reconstruction error of the training faces can actually reach zero, when we use all of 415 eigenfaces. This however cannot be said about the test faces.

Similarly, we observe that the distortion is monotonously decreasing as we increase number of the PCA bases. This is expected as, with each extra PCA base used to reconstruct we take away the biggest eigenvalue from the total sum.

3.2. Nearest Neighbour (NN) classification

In order to perform the PCA-based face recognition, the Nearest Neighbour classification method was used. Having calculated the projections of each normalised training face onto the eigenspace, we could take a new image and classify it into a class. This was done by calculating the new image's projection onto the eigenspace, and compare it to the existing projections. The new image was given the class of the projection that it is closest to. The following classifier equation was used to acheive this:

$$e = min_n ||\omega - \omega_n||, \quad n = 1, ..., N \tag{4}$$

where ω is the projection of the new image onto the eigenspace, ω_n is the projection of the training vector x_n and N is the number of training vectors.

Using this method, each testing vector was classified. With the optimum parameter selection, the classification was successful in 70.1293%, and testing images were assigned to the correct class, as shown in Figure 10. However, there were limitations to the classifier. This method works only on how similar the testing image is to one other image, so it is easily fooled by pictures from incorrect classes with similar lighting, or testing images that have a particular difference from the training data, i.e. the removal of the glasses in the failure example in Figure 11.

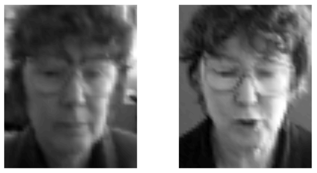

Figure 10. Example of a successful classification: Testing Face, Example from Assigned Class

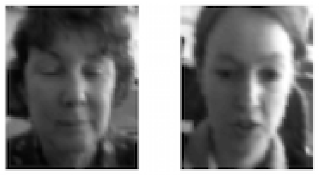

Figure 11. Example of a failed classification: Testing Face, Example from Assigned Class

Varying the number of eigenvectors used to project vectors onto the eigenspace causes the execution time and success rate of the classifier to change, as can be seen in Table 10. It can be clearly seen that as the number of eigenvectors increases, the execution time increases, while the success percentage peaks when the number of eigenvectors is 150.

Num. Eigenvectors	Execution Time /sec	Success %
50	0.227370	63.4615
100	0.232730	68.2692
125	0.241029	69.2308
150	0.245756	70.1923
175	0.263097	69.2308
200	0.292893	69.2308
250	0.357959	68.2692

Table 3. Execution time and success percentage for Nearest Neighbour Classifier with different numbers of eigenvalues

The best case shown in the table above, i.e. when using 150 eigenvectors s presented as a confusion matrix in Figure 12.

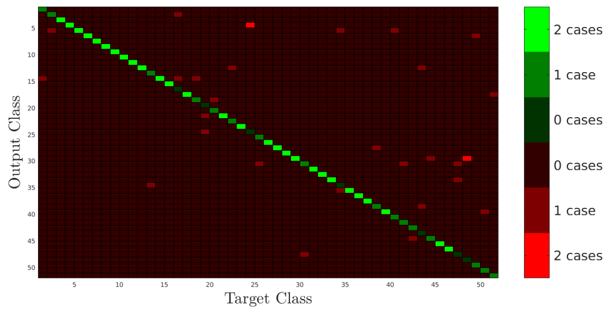

Figure 12. Confusion Matrix of NN method for 150 PCA bases. Green - correct guesses. Red - incorrect guesses.

4. Multi-Class SVM

4.1. Binary Class SVM

The Support Vector Machine in the simple form supports binary classification of data. Hence if we train the classifier with 52 classes we will obtain meaningless results. This section is based on the LibSVM Matlab library provided by [3]. In order to achieve multiclass SVM classification we can implement one of two algorithms: one-versus-one (OVOSVM) or one-versus-all (OVASVM). These two are both based on the binary version of SVM. There are other extensions of SVM which also allow multiclass recognition.

4.2. One-versus-one SVM

For C number of classes, one-versus-one SVM technique requires $T = {}^{(C-1)}\!/_2 \times C$ trainings and testings as we want to compare each pair of classes and run the test image past the resulting kernels. It can be seen already that this method is computationally expensive. In our case $C = 52$ and so $T = 1325$. However each of the kernels is relativley easy to compute as it is made up of 16 training points, due to our data division.

The test image is passed through all 1325 kernels. Each kernel will assign the image to class j or i, $1 < j < C \quad j < i < C + 1$. Those are called votes. For instance if we input an image of class 3, we expect (ideally) that all 51 kernels which are based on class 3 will result in 3 being assigned to the image. We then expect that all other kernels will 'randomly' assign other classes, thus making class 3 the highest voted class.

4.3. One-versus-all SVM

Another method of performing the multiclass SVM classification is to employ the one-versus-all SVM method. Instead of creating 1325 kernels, we only require 52. By taking class j as one part of the kernel, the OVASVM algorithm essentially assumes that the remaining $C - 1$ classes are just one class. That way we can examine if the image belongs to class j or the rest. Ideally OVASVM should result in +1 being outputted just once. This assumption could allow us to, on average, halve the computation time, as we wouldn't have to check other kernels if we have found the correct class early on. However this time in order to compute the kernel, we employ all 416 training images, which takes considerably longer.

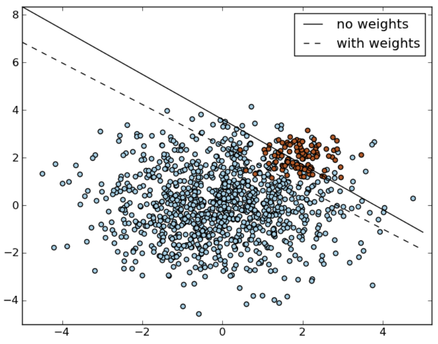

Figure 13. Example of unbalanced binary class SVM [4]

However, OVASVM's disadvantage is that the classes are no longer balanced. This means that one class contains more data than the other. This affects the way the hyperplane is constructed. Practically, this means that the minority class would be ignored as shown in Figure 13. This can be compensated for with changing the way misclassification is penalised - commonly known as class-weighted SVM. Using the LibSVM library we do it by inserting the `-wi` flag.

4.4. Discussion

The two SVM methods were tested for pixel intensity and PCA components. Each classification approach was tested by varying the kernel settings. The results are shown in Tables 4 and 5. Note that 'Acc' means success rate, Time$_{Tot}$ shows total execution time for 104 test images and Time$_{Succ}$ represents time per successful identification.

Table 4. Multiclass SVM for Raw Pixel Intensity Vectors.

Method	**Kernel**	**Acc. %**	**Time$_{Tot}$ s**	**Time$_{Succ}$ s**
OVO	linear	96.2	325.5	3.3
OVO	quadratic	96.2	295.9	3.0
OVO	radial	33.7	379.1	10.8
OVO	sigmoid	1.9	370.5	185.3
OVA	unweighted linear	98.1	624.3	6.1
OVA	weighted linear	98.1	631.7	6.2
OVA	unweighted quadratic	94.2	640.2	6.5

Table 5. Multiclass SVM for PCA bases.

Method	**Kernel**	**Acc. %**	**Time$_{Tot}$ s**	**Time$_{Succ}$ s**
OVO	linear	96.2	31.1	0.31
OVO	quadratic	74.0	31.3	0.41
OVO	radial	1.9	32.6	16.3
OVO	sigmoid	7.7	21.3	2.7
OVA	unweighted linear	96.2	38.3	0.38
OVA	weighted linear	96.2	38.4	0.38
OVA	unweighted quadratic	84.6	55.2	0.63

We can examine the various approaches in multiple ways. First of all just by comparing the success rates we see that linear OVOSVM and both linear version of OVASVM produce nearly equally accurate results varying between

96% and 98%.

However we should also take into account how efficient a given algorithm is. Even though linear kernels in OVOSVM and OVASVM produce the same results, OVA takes significantly longer to execute. This is due to the fact that each time we are training the SVM classifier using all 416 vectors. For 104 testing images we train the SVM a total of 5408 times. This takes much longer than OVO, where SVM is trained 137904 times. However each training process uses only 16 images, thus making it very fast. Therefore the fastest method has been determined to be OVO with a sigmoid kernel for PCA bases.

This raises an interesting point. We can use the dimensionally reduced images, i.e. the PCA bases for face recognition. Instead of using 2576 features, the algorithm now only employs top 150 varying features. Thus making the process on average 13.3 times faster.

Finally, we can look at the combined measure of efficiency and accuracy - time per successful identification. This has been calculated by simply dividing the total execution time by the number of successful recognitions. The clear winners are again the linear kernel, since they are the simplest, producing a successful every 310 ms.

Interestingly, the unweighted linear kernel produces the same results as the weighted counterpart, though the expectation was that it would be highly inaccurate. The unweighted binary classes are possibly internally corrected by the `svmtrain` function, despite the fact that `-wi` flag is not set.

Table 6 presents average training and testing times for OVO and OVA with PCA and full images.

Table 6. OVO and OVA training and testing times

Method	Kernel	Test/Train	Data Type	Time [s]
OVO	linear	Train	PCA	97μ
OVO	linear	Train	Images	2m
OVO	linear	Test	PCA	30μ
OVO	linear	Test	Images	417μ
OVA	linear	Train	PCA	8m
OVA	linear	Train	Images	125m
OVA	linear	Test	PCA	96μ
OVA	linear	Test	Images	2.5m

As expected the table supports the argument that OVO method using PCA bases it quicker. OVO training using PCA is roughly 20 times faster then training using full vectors. Similarly testing is about 14 times faster when using PCA.

Thus, the multiclass SVM classification is best achieved with a linear OVO kernel for PCA bases, as it produces 98.1% accuracy rate and is relatively fast, performing 104 classifications in just over 30 seconds. This naturally could be optimised by precalculating all of 1326 models and simply iterating though them rather than having to build them every time a new image comes through.

For completeness Figure 14 shows a confusion matrix of a OVOSVM with a quadratic kernel for PCA bases.

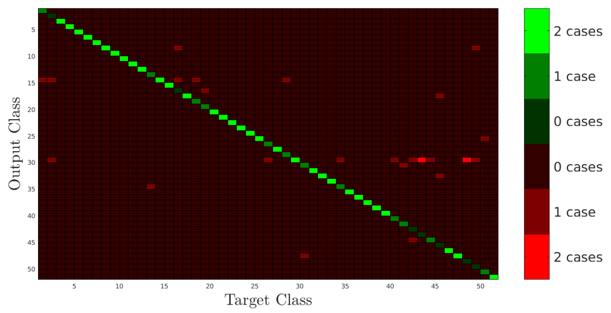

Figure 14. Confusion Matrix of OVOSVM quadratic kernel PCA bases. Green - correct guesses. Red - incorrect guesses.

Finally, Figure 15 shows one of four unsuccessful classifications of the most optimal SVM implementation - SVMOVO with a linear kernel using PCA bases.

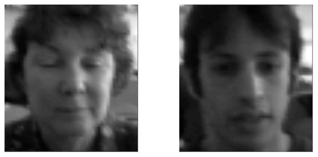

Figure 15. Unsuccessful Classification. Left: Testing Face, Right: Assigned Class

4.5. Comparison of NN and SVM classifiers

The two most striking differences in terms of performance are the execution time and success rates. Firstly, we see that the best implementation of SVM is by far more accurate than optimal NN method, which peaks at 70.2%.

On the other hand the total execution time for NN classification for all 104 test images is in the range of a few hundred milliseconds; 250 ms for 150 PCA bases. This results in average time per successful identification of around 3 ms. For comparison, the best time per successful classification for SVM is around 310 ms, i.e. 100 times longer.

5. Conclusion

This report described Principal Component Analysis, the Nearest Neighbour classifier, and Support Vector Machines and compared versions of each in terms of execution time and classification accuracy. We also found that an SVM can be more accurate than a Nearest Neighbour classifier, but has a much greater execution time.

References

[1] Inderjit Dhillon. *CS 391D Data Mining: A Mathematical Perspective Fall 2009*. The University of Texas at Austin, September 2009.

[2] Victor Y.Pan, Zhao Q. Chen, Ailong Zheng. *The Complexity of the Algebraic Eigenproblem*. Lehman College and Graduate Center, CUNY, NY, December 1998

[3] Chang, Chih-Chung and Lin, Chih-Jen *LIBSVM: A library for support vector machines* ACM Transactions on Intelligent Systems and Technology, vol. 2, iss. 3, 2011, 27:1–27:27 Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`

[4] D. Cournapeau *3.2. Support Vector Machines* scikits.learn developers (BSD License), 2010

6. Matlab Scripts

6.1. OVOSVM

```
  function [assignedClass] =
OVOSVM(testingImage, testLabel, training)

  % One vs One SVM requires us to carry
out (for classSize = 52) 51+50+49...+1
trainings -> that is T = (classSize -
1)*classSize/2 trainings.  Then a testing
face is fed into each of T models and
votes are counted.  Each model will
return a value class1, class2 -> number
of occurrences will be counted.  The one
with biggest amount of votes will be the
winner

  votes = zeros(1,52);
for i = 1:51
  for j = i+1:52

  % train each set of classes once -> 1 v 2
== 2 v 1
    class1 = i;
    class2 = j;

  % extract the right data for easier
handling
    binaryTrain = [training(:,(class1-1)*8+1
:  ( class1 -1) *8 +8)
training(:,(class2-1)*8+1:(class2-1)*8+8)]';
    trainFlags = [class1*ones(1,8)
class2*ones(1,8)]';

  % estimate model for those two classes

    SVMModel = svmtrain(trainFlags,binaryTrain
,'-t 0 -q');

    [label, , ] = svmpredict(testLabel,
testingImage, SVMModel, '-q');

    votes(label) = votes(label) + 1;

  end
end

  [ , assignedClass] = max(votes);
end
```

6.2. OVASVM

```
  function [assignedClass] =
OVASVM(testingImage, testLabel, training)
scores = zeros(52,1);
for i = 1:52
class1 = i;

  trainFlags = -ones(size(training,2),1);
trainFlags( (class1-1)*8+1:(class1-1)*8+8 )
= 1;

  SVMModel = svmtrain(trainFlags,
training', '-t 0 -q');

  [ , ,scores(i)] = svmpredict(testLabel,
testingImage, SVMModel,'-q');

  end

  [ , assignedClass] = max(scores);
end
```