Pattern Recognition Coursework 1

Theo Franquet
00967383
tcf14@ic.ac.uk

Guillaume Ramé
00978741
gr1714@ic.ac.uk

1. Question 1 - Principle Component Analysis

1.1. Eigenfaces

1.1.1 Partitioning the data

Data is partitioned into a training set (70%), a validation set (10%), and a testing set (20%). The method used to form our subsets is stratified sampling, where the train set contain faces of all classes present in the sample set. Using such method leads to having classes present in the validation and testing sets included in the training set, and is crucial in order to better evaluate the performance of Principle Component Analysis (PCA) when reconstructing faces in the test set. The training set T is then zero-centered by subtracting the mean face $\bar{\mathbf{x}}$ (Figure 1) in order to form A, the *centered training set* which has dimensions 2576×364 (2576 pixels & 364 faces):

$$A_i = T_i - \bar{\mathbf{x}}$$

for the i^{th} image. Here, extracting the mean face from our training set is equivalent to removing the face features common to all classes that won't have much influence on class separation. For the rest of this paper, we assign D to the number of features ($D = 2576$) and N to the number of images available for training ($N = 364$).
Each pixel of $\bar{\mathbf{x}}$ is the mean of the respective pixel for all faces of T.

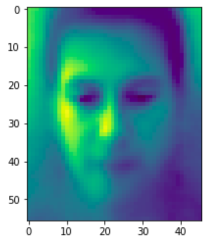


Figure 1. Mean Face

1.1.2 Covariance matrix using AA^T

The covariance matrix S_1 is computed using:

$$S_1 = \frac{1}{N} AA^T$$

Note that the resulting covariance matrix is of dimension $D \times D$. We then compute the 2576 eigenvalues and eigenvectors of S_1 using eigen-decomposition and sort them in decreasing order. Large eigenvalues account for the most data variance in the set of face images. All eigenvectors are orthonormal and the best M eigenfaces $u_1, u_2, , u_M$ span the M-dimensional subspace called face space of all possible images. On the other hand, eigenvectors corresponding to eigenvalues close to zero hold very little information about our training set. We can observe this phenomenon in Figure 2, where the 1^{st} eigenface is compared to the 300^{th} eigenface. The former includes important face features such as the eyes, nose and mouth, whilst the latter is very noisy and accounts for little to no face features. In our case, 2213 of all 2576 eigenvalues are below 10^{-10}, representing 86% of all eigenfaces. The last non zero eigenvalue is the 363^{th}, this is due to the fact that only 364 images are available for training ($N - 1$ axises where data varies). Figure 3 illustrates the disparities relating to the importance of eigenvalues, showing the cumulative sum of all normalized eigenvalues from our ordered eigenvalues vector. Looking at Figure 3, only **114 eigenvalues are needed** to account for 95% of the total variance. These results suggest that only a limited number of eigenvectors are necessary to obtain reasonable results for face reconstruction and recognition (114 in this case).

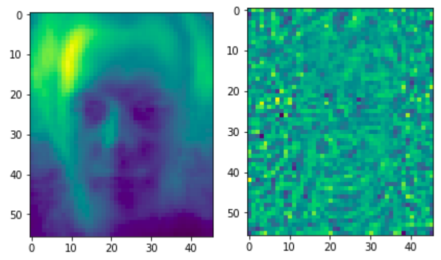


Figure 2. 1^{st} eigenface vs. 300^{th} eigenface

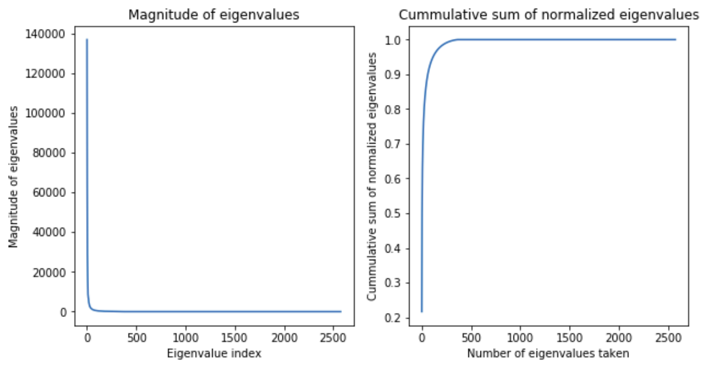


Figure 3. Magnitude of Eigenvalues and Cumulative Sum of Normalized Eigenvalues

1.1.3 Covariance matrix using $A^T A$

Next, we look at the covariance matrix S_2:

$$S_2 = \frac{1}{N} A^T A$$

In this case, the matrix S_2 has dimension $N \times N$. We are looking to compare the eigenvalues and eigenvectors of both S_1 and S_2. Before computing anything, the dimensions of S_2 tell us that its eigenvectors v_i will only have a dimension of 364, and hence cannot be used for image reconstruction. The relationship between the eigenvectors u_i of S_1 and v_i of S_2 is: $u_i = Av_i$ (proven in lectures, $||u_i|| = 1$).

We have just shown that the eigenvectors of S_1 can be retrieved using the ones of S_2. Note however that $i \in [1 : N]$ and hence only 364 of the eigenvectors can be recovered. In order to confirm that **all** eigenvalues of S_2 are equal to the first N of S_1 (mathematical proof in Appendix A), we compute the maximum element-wise difference between the two eigenvalue vectors (taking only the 364 first elements for S_1). Regarding eigenvectors, we similarly compute the maximum mean of element-wise differences between u_i (obtained from S_1) and Av_i (obtained from S_2).

Initial results show unexpected high error for eigenvectors. Looking into the values of both eigenvector matrices, we notice that the only differences are that some eigenvectors are oriented in opposite directions (eigenvectors multiplied by the scalar -1). This means that although the eigenvectors are in fact different, the overall basis formed by all eigenvectors remains identical. Multiplying each eigenvector by the sign of its first element, we obtain near zero differences, confirming our hypothesis:

```
Max eigenvalue difference: 3.940253000635913e-11
Max eigenvector mean difference: 7.98168280153013e-15
```

Figure 4. True error between S_1 and S_2 eigenvalues/eigenvectors

1.1.4 Method Comparison

Table 1 highlights the pros and cons of both methods seen above. When applying PCA, most of the computational load comes from finding eigenvalues and eigenvectors. Finding eigenfaces using $A^T A$ greatly improves the execution speed of the program as only $\frac{1}{7}$ of all eigenvalues and eigenvectors have to be found (relative to AA^T) ($D >> N$). As such, memory usage is also reduced by a factor of $\frac{2576^2}{364^2} = 50$. This highly efficient method will be used subsequently to perform image reconstruction and face recognition.

It is however extremely important to note that the second method is only more efficient in the case that the size N of the training set is much smaller than D the dimensionality of the faces in pixel space. In the case that $N = D$ or even $N >> D$ it would be preferable to use the technique introduced in section 1.1.2 to set an upper bound of D on the size of the covariance matrix / eigenvector matrix. This also prevents from having to compute $\mathbf{u} = A\mathbf{v}$ which will be very expensive for large datasets.

1.2. Application of eigenfaces

1.2.1 Reconstruction

Reconstruction is executed using a linear combination of all eigenfaces (seen in lecture notes):

$$\tilde{\mathbf{x}}_n = \bar{\mathbf{x}} + \sum_{i=1}^{M} a_{ni}\mathbf{u}_i$$

Looking at this formula, increasing the number of eigenfaces used should improve the reconstruction accuracy, as more more bases are available (large M). However, as seen previously, eigenfaces corresponding to small eigenvalues account for negligible data variance and can be ignored. To better observe both these phenomenons, PCA was applied on three images of the training set as well as from the testing set, each time using a different number M of PCs (see Figures 5 & 16).

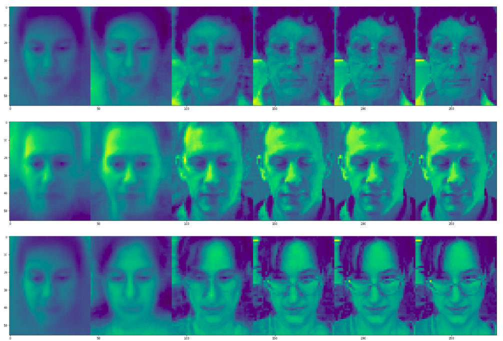


Figure 5. Training set reconstruction: from left to right, $M = 2; 10; 100; 200; 300$. Right-most is the original image

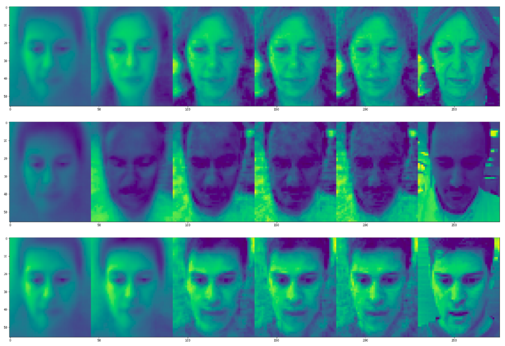


Figure 6. Testing set reconstruction: from left to right, $M = 2; 10; 100; 200; 300$. Right-most is the original image

We can observe two things: That increasing M early on drastically improves the overall quality of the reconstruction. And that reconstruction quality does not significantly improve beyond $M = 100$.

The quality of the test set reconstruction drops slightly and is substantially noisier and grainier. This shows that the PCA bases learned are overfit on the training set, as it is clear from the test images that these bases do not capture some of the variation present. As a consequence, we would have benefited from more training data. Interestingly, the reconstruction for the darker skinned middle person seems also much worse than that of the other classes. A possible explanation could be that the train set simply does not contain enough variation of non-white classes in order for good appropriate bases to be learned. This is a common issue with publicly available datasets, especially considering that a large number of these exclusively consist of mostly white classes.

Theoretical and experimental reconstruction error as a function of M is shown in Figure 7, for both the training and the testing set.

Theoretical reconstruction error J_t is obtained by:

$$J_t = \sum_{n=M+1}^{D} \lambda_n$$

	AA^T	A^TA
Pros	-	smaller dimensional data: low memory fast eigenvalues/eigenvectors computation all important eigenfaces can be obtained
Cons	larger covariance matrix: memory intensive more eigenvalues and eigenvectors to compute low overall efficiency	extra computational step ($u_i = Av_i$) to recover faces

Table 1. Comparison Table

Experimental reconstruction error J_e is obtained by:

$$J_e = \frac{1}{N}\sum_{n=1}^{N}\|\mathbf{x}_n - \tilde{\mathbf{x}}_n\|^2$$

As expected, both the theoretical and experimental error quickly converge to zero as M increases. Both J_t and J_e follow the same trend, more importantly: J_t matches with the training J_e as both errors rely on the same eigenvalues and face images. This phenomenon can be explained by looking back at Figure 3: the magnitude of eigenvalues is directly correlated to how much variance is taken into account by its corresponding eigenface.

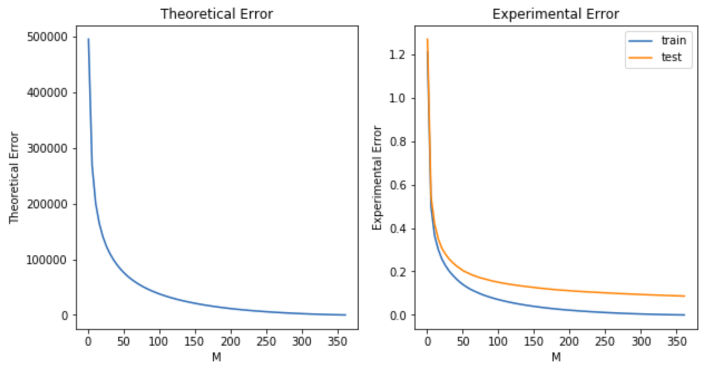


Figure 7. Theoretical and experimental error with increasing face space dimensionality

1.2.2 PCA-based face recognition

PCA-based face recognition was accomplished using the nearest neighbors classification method. The first step was to find a new number m of eigenvectors which should be used for image recognition. This was accomplished by performing PCA-based face recognition using increasing values of m on our validation set. Results are shown in Figure 8. Maximum accuracy (around 54%) is reached as soon as $m = 121$, which approximatively corresponds to our estimation done in section 1.1.1.

Measuring testing accuracy gives a result of 51%. An example of confusion matrix is shown in Figure 9.

An example of success and failure cases is given in Appendix C for a specific face. Looking at the images that failed to be correctly labeled, we notice that the light intensity, face expression, and photograph angle is different to the correctly labeled image of the same class. These features were not necessarily taken into consideration during training and image reconstruction of these images might make those particular faces resemble faces of a completely different class. An example of this can be seen in Figure 10. We

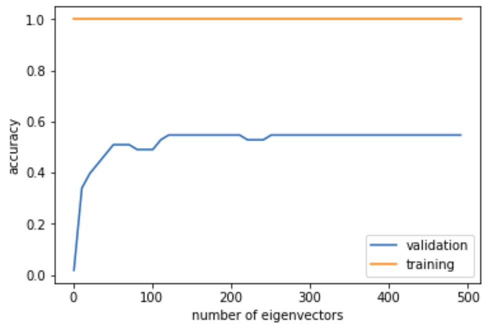


Figure 8. Validation and training accuracy with increasing values of m

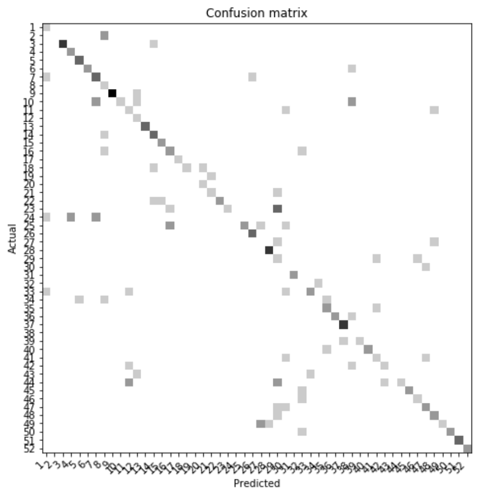


Figure 9. Test Set Confusion Matrix

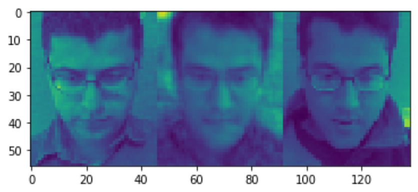


Figure 10. Left: Training face from class 48 with projection closest to middle face from class 27. Middle: Face from class 27 in test set. Right: Train image from class 27 closest to middle image in PCA space.

Exec. Time (S_1)	Exec. Time (S_2)
4.75s	0.88s
Memory (S_1)	Memory (S_2)
2576^2	364^2

Table 2. Execution time and memory usage

can see that the poor quality of the reconstruction amounts to a face that has extremely similar facial features to the face to the left (class 48). This combined with the identical pose and lighting conditions in the original image (right, class 27) results in a misclassification. By manually calculating the euclidean distance between the projection of classes 27 and 48, we can see that the middle face is slightly closer to the projection from 48 than that of class 27 (distance of 1185 vs distance of 1213).

We are also interested in looking at the impact of training set size on face recognition accuracy and execution speed. The hypothesis is that accuracy and execution time increase with increasing train set size. Experimentally, we look at both metrics, each time forming new randomly generated training sets based on the initial $D \times N$ training set, before testing on our validation set. Results shown on Figure 11 suggest that the hypothesis is true, which seems to confirm the statement that a larger sample size would have improved recognition accuracy, at the cost of computational complexity.

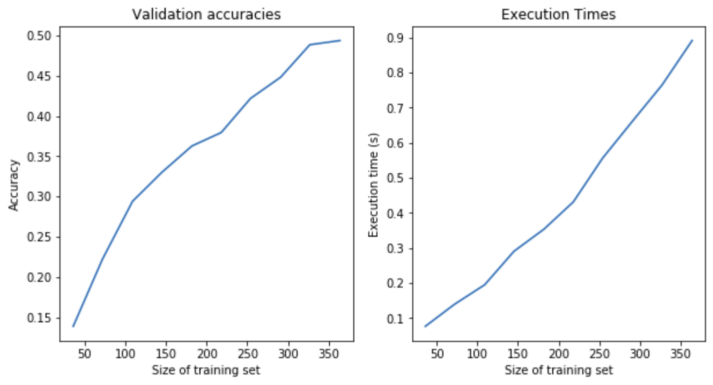


Figure 11. Validation accuracies and execution times vs. Size of training set

The execution time and memory usage of our program has been monitored using both AA^T and A^TA methods. In our case where $D >> N$, using A^TA results in lower covariance matrix dimensionality, leading to shorter computation times and lower memory usage. Results are shown in Table 2. As expected, using S_2 results in lower execution times ($5.4\times$ faster) and memory usage ($50\times$ lower). Here, execution times refer to the time necessary to extract eigenfaces and apply NN-based face recognition on the validation set.

2. Question 2 - Support Vector Machine

2.1. Data preprocessing

Similarly to the first question, the dataset is partitioned into a train / validation sets (80%) and a testing set (20%). As we are aiming for closed-set face recognition, we also make sure that the classes present in the validation and test sets are a subset of those present in the training set. As per best practices, only the validation set will be used for hyper-parameter tuning. A trained classifier's accuracy on the test set will then only be evaluated once, once the necessary tuning has been done. The coefficients used for the PCA test were computed using *SciKit-Learn's PCA* object. The dimensionality of the coefficient vectors (approx. 125) was determined such that 95% of the data's total variance was maintained. The data fed into PCA and the SVM (including the test set) is normalized using a mean and standard deviation computed solely on the training set.

2.2. Multi-class SVM

Support Vector Machines (SVM) are, by nature, only capable of binary classification. That is, a single classifier is only able to distinguish two classes of points. This means that the base algorithm must be extended in order to be used for multi-class classification. This can be achieved using two methods: one versus one and one versus rest classification schemes. As shown in Figures 21 and 22 (Appendix D), the code implementing both these algorithms is encapsulated in a convenient to use python class, adapted to fit the *SciKit-Learn Estimator* interface, allowing one to execute parallel cross-validation, using the *GridSearchCV* object from *sklearn.model_selection*.

2.2.1 One vs. One (OvO) classification

In one versus one classification, a binary classifier is trained per pair of classes from all possible combinations of classes in the dataset. This amounts to training $n(n-1)/2$ binary classifiers, where n is the number of classes. This leads to the evaluation of $n(n-1)/2$ classifiers in the prediction stage. These predictions are then reduced per point by majority vote.

2.2.2 One vs. Rest (OvR) classification

In one versus rest classification, a binary classifier is trained for every class in the dataset. Each selected class is in turn selected as the positively labeled class, while all the remaining classes are set to be negative examples. This method produces n classifiers, where n is the number of classes in the dataset. In the prediction stage, all binary classifiers are evaluated and the $argmax$ of the decision function values determines the label for a given test point.

2.3. SVM hyper-parameter tuning

Support Vector Machine have a number of hyper parameters, including the type of kernel used, the box constraint and parameters specific to certain kernel functions. The optimal parameter for any given train / test split will be determined using 3-fold cross-validation. The cross-validation python object mentioned in 2.2 internally generates an appropriate number of train / validation splits according to a user-specified number (defaulting to 3 used here). It will then average the validation accuracy over the number of folds, and compute this for all combinations of parameters.The validation accuracies produced in the following sections were extracted using the *best_score_* member variable, which holds the cross-validation accuracy for the model using the best found hyper-parameters. The kernel type used in the following tests consisted of Linear and Radial Basis Function (RBF) kernels. The box-constraint C, adjusting how much soft-margins are penalized (higher C meaning harder margin), will be varied in log-increments from 10^0 to 10^3. $gamma$, used in the RBF kernel, will be varied similarly from 10^{-7} to 10^{-1}. The grid of values and number of cross-validation folds was kept relatively coarse and low to reduce the runtime of the experiments.

2.4. Results

2.4.1 Recognition Accuracy

Table 3 shows mean validation accuracy averaged over the three cross-validation folds for the best performing classifier for any given combination of kernel type, usage of PCA and classification scheme.

Type	PCA	Kernel	Val. Acc.	Opt. Parameters
One v Rest	no	RBF	81.0%	$C : 1000, \gamma : 10^{-5}$
	no	Linear	81.7%	$C : 1$
	yes	RBF	81.5%	$C : 1000, \gamma : 10^{-5}$
	yes	**Linear**	**81.7%**	$C : 1$
One v One	no	RBF	66.8%	$C : 1000, \gamma : 10^{-7}$
	no	Linear	66.8%	$C : 1$
	yes	RBF	66.1%	$C : 100, \gamma : 10^{-5}$
	yes	Linear	66.6%	$C : 1$

Table 3. Validation Accuracy for varying parameters

In these results, one versus rest classification emerges as clearly superior to one versus one, most notably in terms of validation accuracy. For both pixel and PCA cases, the linear kernel has a slight validation advantage over the RBF kernel. In the end, the linear kernel is selected as it also proves to be marginally faster in inference phase and substantially faster in training as grid search only needs to sweep the box-constraint. The linear kernel is also likely to scale better with larger dataset sizes since parts of its decision function can be precomputed. OvR's vastly better recognition accuracy could be caused by the fact that while each binary classifier in the OvR case has a global view of the data, each binary classifier in the OvO case only has a very local view of the feature space. This means that the OvR classifier can effectively ensure that the margin for each class is maximized with respect to the rest of the dataset, which by consequence, allows OvR to generalize much better than its counterpart.
Interestingly, in our experiments, cross-validation for the models using RBF kernels returns very small optimal values for $gamma$ (usually $< 10^{-5}$). Intuitively, the $gamma$ parameter tunes the influence of the support vectors on the decision boundary. If gamma is very high, the decision boundary will have a tendency to curve to meet the support vectors (as seen in Figure 19), usually leading to over-fitting. In contrast, small $gamma$ will make the RBF kernel behave similarly to a linear kernel (as seen in Figure 18), possibly leading to under-fitting. We can see from the above results that the linear kernel and RBF kernel have similar accuracy, explaining why very small values of $gamma$ are optimal. This coupled with the fact that the box-constraint is usually larger than 10 shows that the dataset is quite linearly separable, both in pixel and PCA spaces, with small amounts of margin violation. This is compounded by the fact that the data's already high dimensionality when compared to the sample size means that there is little benefit in the kernel mapping it to a higher dimensionality than it already is.

Figure 12 shows the confusion matrix for the test set inferred using a model using parameters cross-validated with the methodology introduced previously. The final parameters were: $C = 1$, using a linear kernel with one versus rest classification and PCA enabled. Before the evaluation of the test accuracy, a model is retrained from scratch using the entire training set. This results in a 89.4% accuracy on a test set of size 104. The higher test accuracy of the SVM when compared to the nearest neighbor approach used in 1.2 is visually obvious from the increased density of the matrix's diagonal.

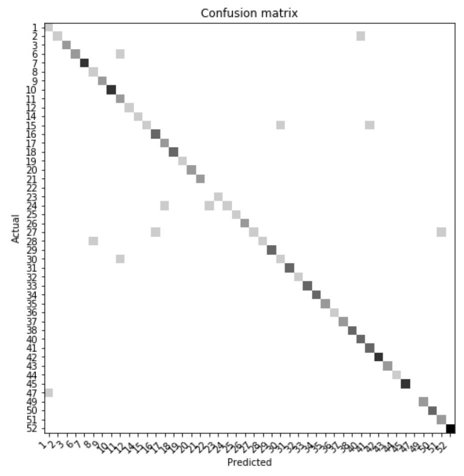


Figure 12. Test Set Confusion Matrix

2.4.2 Time efficiency of SVM training/testing

The following part demonstrates that there are large variations in computational efficiency when comparing various forms of the SVM classifier. Table 4 shows that using PCA coefficients rather than raw images brings a large speedup in terms of mean time for a 3-fold cross-validation by more than an order of magnitude. The values in this table were computed by scaling the total time for an entire cross-validation run by the number of total parameter combinations. As seen in 5, using PCA also translates into the substantial speedup at testing time. Data points for 5 were generated by inferring batches of uniform noise arrays - of equal dimensionality to the training set - with various trained classifiers. These batches have size n, size of the training set (416 in our case).

Tables 4 and 5 both show a substantial reduction in performance when classification scheme is switched to OvO. While the OvO training phase doesn't seem to be much slower than that of OvR, this comes from the fact that the large increase in the number of SVMs to train (from n in OvR to $n(n-1)/2$ in OVO) is compensated by the reduction of the amount of data given to each SVM.This advantage is completely removed in the testing phase, where all $n(n-1)/2$ SVM's need to be inferred on the entire test set. This ultimately means that inference of OvO in this test is approximately an order of magnitude slower than OvR. The inherent difference in theoretical complexity between the two methods means that this gap in train and test times will continue to increase as the number of classes and average number of points per class is increased.

Type	PCA	Kernel	Time per train iteration
One v Rest	no	RBF	12.3 s
	no	Linear	4.0 s
	yes	RBF	0.48 s
	yes	Linear	0.54 s
One v One	no	RBF	9.1 s
	no	Linear	10.7 s
	yes	RBF	1.0 s
	yes	Linear	1.3 s

Table 4. Train time as a function of algorithm, kernel and PCA.

Type	PCA	Kernel	Time per inference
One v Rest	no	RBF	3.24 s
	no	Linear	3.08 s
	yes	RBF	0.12 s
	yes	Linear	0.11 s
One v One	no	RBF	20.78 s
	no	Linear	21.67 s
	yes	RBF	1.29 s
	yes	Linear	0.93 s

Table 5. Total test inference time as a function of algorithm, kernel and PCA.

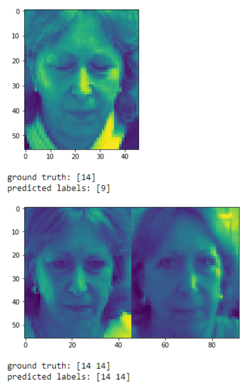
2.4.3 Recognition examples

Figure 13 shows examples of correctly classified and misclassified test examples. When compared with Figure 16, depicting a similar test but applied to a nearest neighbor classifier, we can see that the former seems more robust to pose, expression and lighting variations than the latter. This is in agreement with the substantially improved test set accuracy shown previously. The misclassified example shows that the classifier is however still prone to error when a test image is quite different to that present in the training set (shown in Figure 14). In this case, closed eyes and what seems to be slightly different lighting conditions is enough to cause a misclassification.

Figure 14 represents class 14 of the dataset along with the support vectors of its corresponding one versus rest trained SVM. For this specific class, there are a total of 37 support vectors from 18 different classes. The fact that the SVM uses far less support vectors than there are data points available in the training set shows that the model is able to take advantage of the apparently localized structure of the dataset in the PCA feature space. This phenomenon is not an artifact of this class as each class uses 34.1 support vectors on average. As expected, the support vectors are a mix-and-match of class images and what are likely to be the *closest* images in the PCA space.

2.5. Reflection

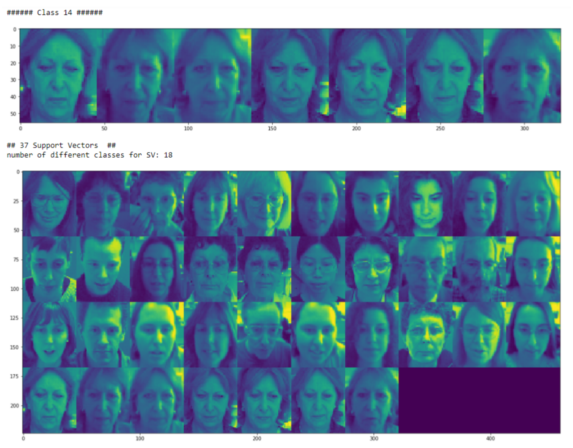
When compared with the results in the first exercise, the validation and test accuracies are significantly better when the nearest neighbor classifier is switched out for an SVM. On top of this, there is also substantial improvement in validation accuracy when comparing one versus rest and one versus one classification schemes. A possible reason as to why the SVM could be outperforming the nearest neighbor model by such a large margin could be that the SVM is inherently less sensitive to a noisy dataset or outliers. This is especially an issue when working with data with very high dimensionality such as images, which are very prone to large amounts of noise. By maximizing the margin between a positive and negative class, SVMs are naturally able to abstract out some of the noise in the dataset, leading to better recognition performance. A SVM classifier is also likely to scale substantially better as the size of the training set is increased. This comes down to the fact that a nearest neighbor classifier, while being *lazy* and not requiring training *per se*, requires that the training set be stored as it is needed for inference. Inference with a nearest neighbor also requires the evaluation of the distance metric between each test point and the entirety of the train set. In contrast, each SVM for a multi-class SVM needs to only store its relevant support vectors, and inference is very efficient as it only needs to compute the kernel function for support vectors as the dual coefficients for non-support-vectors are zero regardless.



```
ground truth: [14]
predicted labels: [9]
```

```
ground truth: [14 14]
predicted labels: [14 14]
```

Figure 13. Examples of correctly classified and misclassified test points using an SVM



```
###### Class 14 ######
```

```
## 37 Support Vectors  ##
number of different classes for SV: 18
```

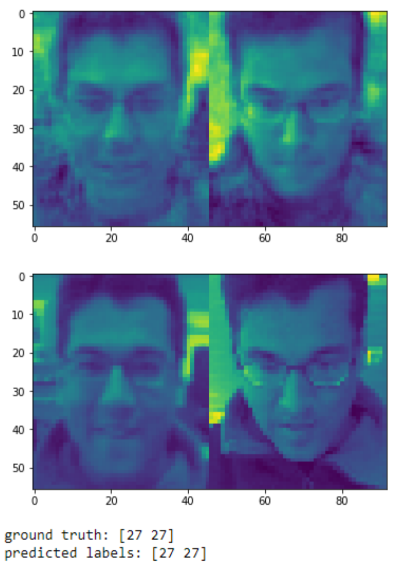
Figure 14. Class 14 and its support vectors

Appendices

A. Non-zero eigenvalues equivalence between AA^T and A^TA (proof)

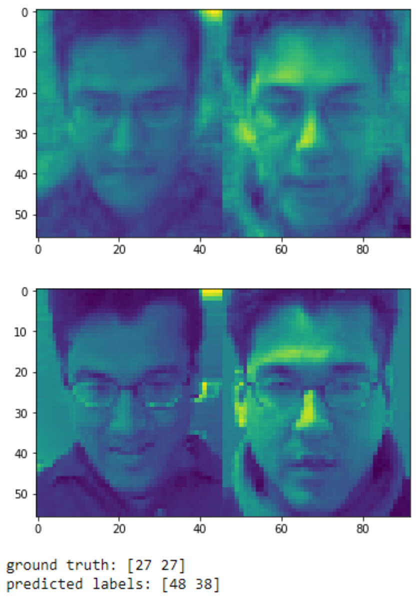
Let A be an $n \times m$ matrix: $A^TAv = \lambda v$
$\implies AA^TAv = A\lambda v = \lambda Av$
$\implies AA^Tu = \lambda u$
for $u = Av$

B. Face recognition success and failure cases (PCA)



```
ground truth: [27 27]
predicted labels: [27 27]
```

Figure 15. Example of successfully identified faces and corresponding projections during testing



```
ground truth: [27 27]
predicted labels: [48 38]
```

Figure 16. Example of unsuccessful identification and corresponding projections during testing

C. Toy examples of SVM classification

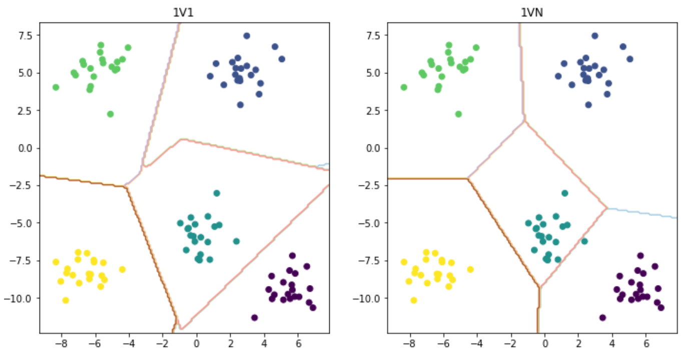


Figure 17. Classification of toy dataset using linear kernel and generic parameters

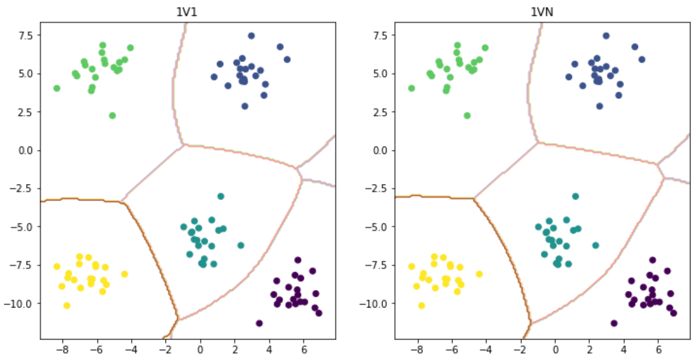


Figure 18. Classification of toy dataset using RBF kernel and generic parameters, $\gamma = 0.1$

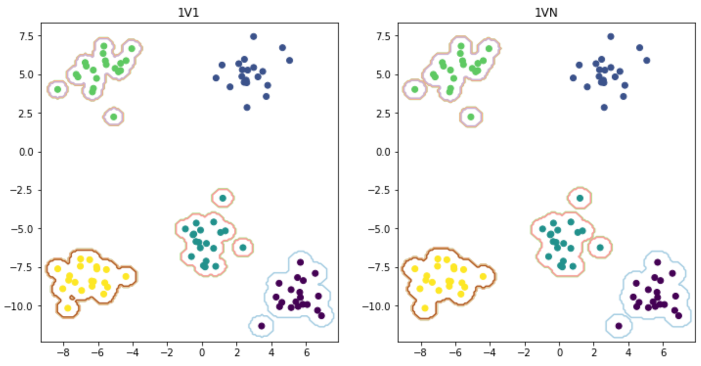


Figure 19. Classification of toy dataset using RBF kernel and generic parameters, $\gamma = 10$

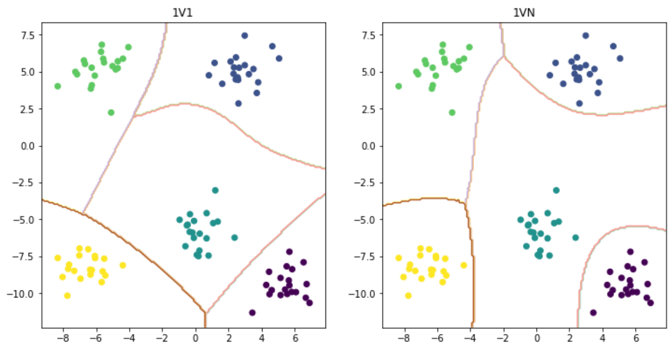
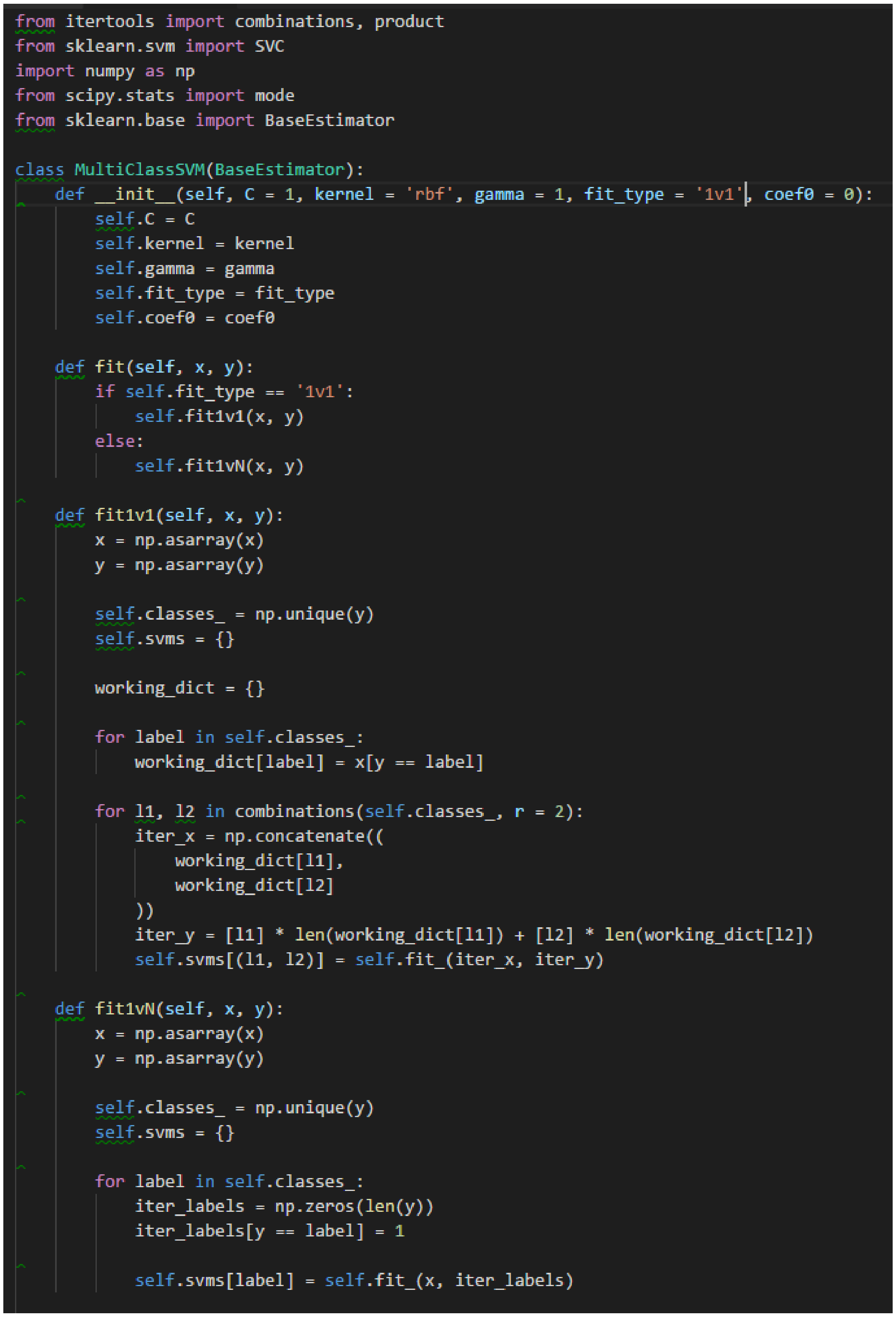


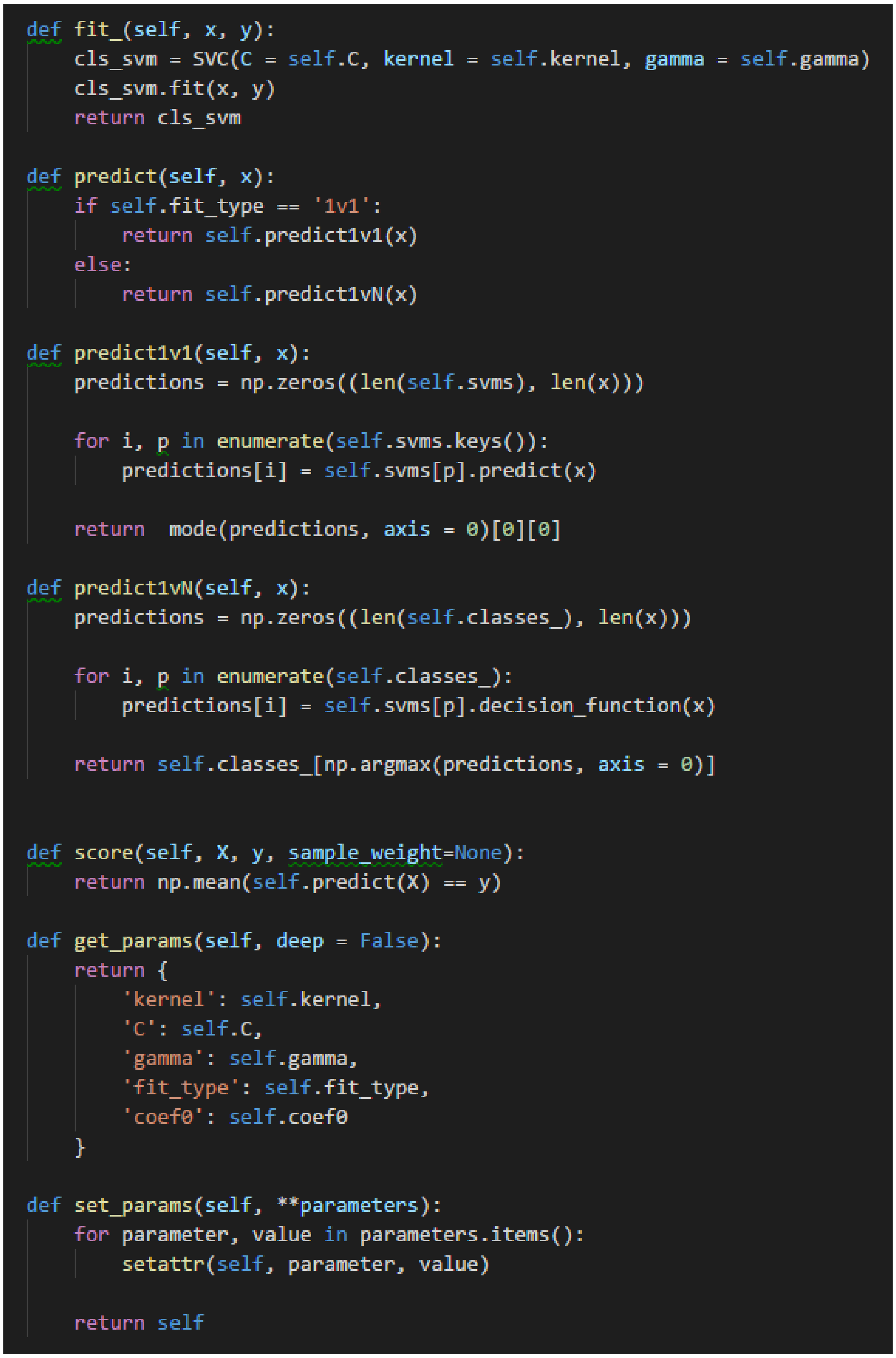
Figure 20. Classification of toy dataset using polynomial kernel and generic parameters

D. Code for multiclass SVM implementation



```python
from itertools import combinations, product
from sklearn.svm import SVC
import numpy as np
from scipy.stats import mode
from sklearn.base import BaseEstimator

class MultiClassSVM(BaseEstimator):
    def __init__(self, C = 1, kernel = 'rbf', gamma = 1, fit_type = '1v1', coef0 = 0):
        self.C = C
        self.kernel = kernel
        self.gamma = gamma
        self.fit_type = fit_type
        self.coef0 = coef0

    def fit(self, x, y):
        if self.fit_type == '1v1':
            self.fit1v1(x, y)
        else:
            self.fit1vN(x, y)

    def fit1v1(self, x, y):
        x = np.asarray(x)
        y = np.asarray(y)

        self.classes_ = np.unique(y)
        self.svms = {}

        working_dict = {}

        for label in self.classes_:
            working_dict[label] = x[y == label]

        for l1, l2 in combinations(self.classes_, r = 2):
            iter_x = np.concatenate((
                working_dict[l1],
                working_dict[l2]
            ))
            iter_y = [l1] * len(working_dict[l1]) + [l2] * len(working_dict[l2])
            self.svms[(l1, l2)] = self.fit_(iter_x, iter_y)

    def fit1vN(self, x, y):
        x = np.asarray(x)
        y = np.asarray(y)

        self.classes_ = np.unique(y)
        self.svms = {}

        for label in self.classes_:
            iter_labels = np.zeros(len(y))
            iter_labels[y == label] = 1

            self.svms[label] = self.fit_(x, iter_labels)
```

Figure 21. First half of SVM code



```python
def fit_(self, x, y):
    cls_svm = SVC(C = self.C, kernel = self.kernel, gamma = self.gamma)
    cls_svm.fit(x, y)
    return cls_svm

def predict(self, x):
    if self.fit_type == '1v1':
        return self.predict1v1(x)
    else:
        return self.predict1vN(x)

def predict1v1(self, x):
    predictions = np.zeros((len(self.svms), len(x)))

    for i, p in enumerate(self.svms.keys()):
        predictions[i] = self.svms[p].predict(x)

    return  mode(predictions, axis = 0)[0][0]

def predict1vN(self, x):
    predictions = np.zeros((len(self.classes_), len(x)))

    for i, p in enumerate(self.classes_):
        predictions[i] = self.svms[p].decision_function(x)

    return self.classes_[np.argmax(predictions, axis = 0)]


def score(self, X, y, sample_weight=None):
    return np.mean(self.predict(X) == y)

def get_params(self, deep = False):
    return {
        'kernel': self.kernel,
        'C': self.C,
        'gamma': self.gamma,
        'fit_type': self.fit_type,
        'coef0': self.coef0
    }

def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)

    return self
```

Figure 22. Second half of SVM code