Microsoft Word Author Guidelines for CVPR Proceedings

Byoungjun Kim
KAIST
braian98@kaist.ac.kr

Junbeom Kim
KAIST
dungeon12345@kaist.ac.kr

Q1. PCA

1.1. Eigenfaces

1.1.a

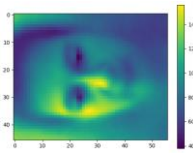


Figure 1 Mean face

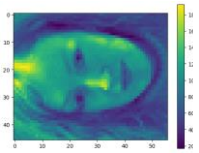


Figure 2 Original PCA

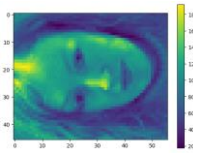


Figure 3 Low-dimensional PCA

Figure 1 show's mean image of dataset. By applying PCA on dataset, got 610 non-zero eigenvalues and it's corresponding eigenvectors. Largest eigenvalue is 929580.62 while smallest is -6.29. Large difference between eigenvalues implies a few eigenvalues and eigenvectors pair who has large eigenvalues plays more significant role for describing eigen-subspace. As shown in Figure 14, eigenvalues drops severely after about 50. Also if we visualize face reconstruction with varying basis from 10, 50, 100, 200, at the point of 50 basis, it starts to get similar shape of origin image in qualitative manner as shown in Figure 16. For eigenvectors, if we visualize eigenvector as in Figure 17 and Figure 18 eigenvector corresponding to larger eigenvalues shows more man-like feature. Therefore, it would be desirable to select at least 50 eigenvectors to be used for face recognition.

1.1.b

Resulting eigenvalues are identical as in 1.1.a. However, number of obtained non-zero eigenvalues differ from 1.1.a which is 519. This difference is occurred since dimension of matrix decreased. For eigenvectors, 1.1.a eigenvectors are linear transformation of 1.1.b eigenvectors by $A(S = \left(\frac{1}{N}\right) A^T A)$. Low-dimensional PCA is more computationally efficient than original PCA original PCA takes 33.8 sec for processing while low-dim PCA takes 1.80 sec, however since it has smaller number of eigenvalues and eigenvectors pair, original PCA can represent more rich feature than low-dimensional PCA in the case original PCA use more eigen pairs.

Figure 2 is reconstructed image by original PCA using 610 eigenvectors and it's reconstruction error is 3.89 and Figure 3 is low-dimensional PCA using 519 eigenvectors and it's reconstruction error is 14.64.

1.2. Application of Eigenfaces

1.2.a

	#Basis	Image1	Image2	Image3	Average
Train	10	1186.80	1294.26	1135.60	1205.56
	50	827.76	871.07	639.01	779.28
	100	611.03	644.28	485.63	580.31
	200	277.92	273.38	344.70	298.67
Test	10	1382.65	1199.30	1228.65	1270.20
	50	1137.58	1001.62	1005.53	1048.24
	100	983.05	868.37	891.19	914.20
	200	877.88	743.96	756.19	792.68

Table 1 Reconstruction error result of train/test images

As in theory, by increasing number of basis, reconstruction error becomes smaller and train image tends to have smaller reconstruction error compare to test image in same condition, since model learned feature from train image as shown in Table 1.

1.2.b

n_components	Accuracy	Time
10	0.5096	0.1937878
50	0.6635	0.2908444
100	0.6731	0.4263844
200	0.6731	0.8906009

Table 2 PCA-based face recognition accuracy varying n_components

Used NN classification method. Table 2 shows that by increasing n_components in PCA, accuracy and time tends to increase linearly. Accuracy increase since it uses more eigenvectors which makes prediction more accurate and leads more calculation for prediction.

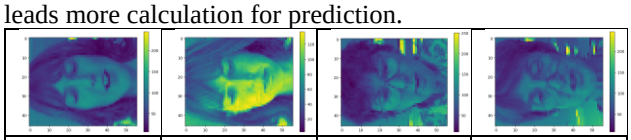


Figure 4 PCA prediction Failure/Success case of n_components=200

As shown in Figure 4, for failure case although it fails to predict ground feature, predict face has a few same feature with ground truth such as wearing glasses, female, middle-aged etc. For success case although luminance varies, it shows model robustly predict within light variance.

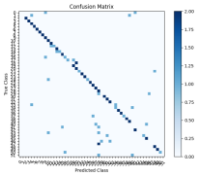


Figure 5 PCA confusion matrix

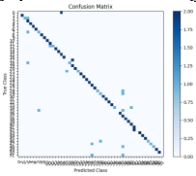


Figure 6 PCA-LDA confusion matrix

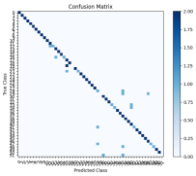


Figure 7 PCA-LDA ensemble confusion matrix

Figure 5 confusion matrix shows that most of data is aligned diagonally which means model performs prediction well most of the time for test set.

Q2. Incremental PCA

Train set	Accuracy	Reconstruction Error	Training time
PCA for 1/4 train set	0.2115	390.04	0.217
PCA batch	0.6730	64.17	0.999
IPCA library	0.6634	74.72	0.881

Table 3 Comparison of variant PCA

Table 3 shows accuracy, reconstruction error and training time of each implementation with NN classification. For 1/4 split train set, since training data is much smaller than the other training can happen in relatively faster however accuracy and reconstruction error is low since train dataset is not enough. IPCA tends train faster than PCA since it incrementally update variable only inspecting smaller train set at iteration. So IPCA train time is similar to 4 times of 1/4 split PCA. However, IPCA update covariance matrix and other variable in assumption, accuracy and reconstruction error indicates lower precision of model compare to PCA.

It is important to choose number of division of training dataset for IPCA. Since IPCA can have n_components up to $\frac{\#\ data\ in\ dataset}{\#\ division}$, so if we increase number of division, training time become smaller but on the other hand accuracy become lower since n_components become smaller which leads to poor performance as shown in Table 2.

Q3. LDA Ensemble for Face Recognition

3.1. PCA-LDA

	M_pca 50	M_pca 100	M_pca 200
M_lda 10	0.769	0.779	0.740
M_lda 20	0.788	0.817	0.798
M_lda 30	0.740	0.798	0.856
M_lda 50	0.663	0.769	0.846

Table 4 Accuracy varying M_pca and M_lda

Table 4 shows accuracy by varying M_pca and M_lda. Accuracy tends to get better when M_pca get larger. However in fixed M_pca, there is saturation point of M_lda. That is because, PCA accuracy is linear with number of eigenvectors but LDA become underfitted since PCA representation has limited dimension which means limited information. Therefore saturation point of M_lda become larger if M_pca become larger which implies rich feature provided for LDA.

$$rank(S_B) \leq \#class - 1$$
$$rank(S_W) \leq \#\ data - \#\ class$$

Rank of the between-class scatter matrix become M_lda and within-class scatter matrix become M_pca. That is because LDA is for classification and we chose M_lda number of eigenvectors for it and chose M_pca number of eigenvectors for PCA to generate within-class.

Figure 6 shows confusion matrix of M_pca is 100 and M_lda is 20.

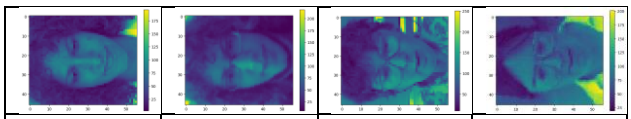


Figure 8 PCA-LDA prediction Failure/Success case

As shown in success case in Figure 8, PCA-LDA can predict two image as same person whether wearing glasses or not. And as shown in confusion matrix it is more diagonally centered compare to Q1. Accuracy becomes better since PCA only consider maximum variance of data while LDA can learn separation between class so that it enhance accuracy of classification.

3.2. PCA-LDA ensemble

# max feature	Accuracy
100	0.942
500	0.923
1000	0.913
2576	0.885

Table 5 Bagging classifier with M_pca=100, M_lda=20

# base model	Accuracy
1	0.875
5	0.894
10	0.885
15	0.885

Table 6 Bagging classifier accuracy varying number of base model

As shown in Table 5, accuracy tends to get lower when number of max feature increase. Increasing number of feature space means considering more features during splitting process which makes model more complex and leads to underfitting.

By randomization on sample data, for # feature space 2576 accuracy become 0.875 which is smaller than without

randomization. Randomization help model to avoid overfit during training, however within small number of dataset, it can leads to underfitting.

Accuracy tends to reach certain point while increasing number of base model as shown in Table 6. It is because increasing number of base model encourage model to become more robust and stable, however after 10 it become mean reversion

Base Model	Accuracy
1	0.702
2	0.702
3	0.702
4	0.789
5	0.712
Committee machine	0.886

Table 7 Error of individual models # base model = 5 random sample data

Fusion rules	Accuracy
Weight voting	0.885
Majority voting	0.885

Table 8 Voting classifier with 5 base model

Table 7 shows that committee machine's accuracy is higher than max accuracy between base models. This is because ensemble enable to reduce overall variance, better generalization which leads to robustness and complement each other. Therefore committee machine accuracy becomes about 0.1 point higher than base model.

Table 8 shows accuracy of voting classifier by varying fusion rules. Difference doesn't quite observable in most of the case since we use same PCA-LDA model for base model. Although weighting specific components, overall accuracy doesn't change severely from PCA-LDA accuracy.

Figure 7 shows confusion matrix of bagging classifier with 5 estimators with random sampling with # feature space is 2576.

Q4. Generative and Discriminative Subspace Learning

In order to obtain a subspace that fulfils both PCA and LDA aspects we have create a newly combined objective function. We utilized the PCA objective as the way it is, which is $J_{PCA}(W) = W^T S_T W$. But we changed the objective function of LDA like below, in order to solve the problem. $J_{LDA}(W) = W^T S_b W - W^T S_w W$. Since LDA's objective function is to increase the between-class scatter while minimizing the within-class scatter, we believe changing the division into subtraction would also act similiarly. Therefore we get combined objective function written below, where alpha is a hyperparameter for two objective function's balance.

$$J(W) = J_{LDA}(W) + \alpha J_{PCA}(W) = W^T S_b W - W^T S_w W + \alpha W^T S_T W$$
$$= W^T (S_b - S_w + \alpha S_T) W$$

Solving the above objective function $w.r.t\, W^T W = I$ is able by using quadratic lagrangian.

$$L = W^T (S_b - S_w + \alpha S_T) W - \lambda (W^T W - I).$$

Calculating derivative of L with respect to w equals 0, we can obtain $(S_b - S_w + \alpha S_T) W = \lambda W$. Which is the definition of eigenvectors and eigenvalues. Therefore, we can obtain W which will map us to our desired subspace by calculating the d eigenvectors of $S_b - S_w + \alpha S_T$ which corresponds to the highest d eigenvalues.

This approach's advantage is that it is simple and efficient to compute since it has a closed form solution. Since it is reducing dimension while utilizing LDA's discriminant ability we believe that it will have more accuracy in classification. The disadvantage of this approach is that is had to approximate LDA's objective function, so when between-class scatter and within-class scatter's scale is different, the objective function might not work well. In that case, we can add another constant β for scaling and obtain adjusted objective function as the below.

$$J(W) = W^T (S_b - \beta S_w + \alpha S_T) W$$

Q5. RF classifier

5.1. Effect of Random Forest Parameters

In this section, we trained and tested random forest classifiers using the given data with a different set of parameters to experiment how they affect the training time, inference time and test accuracy. We utilized the scikit-learn's RandomForestClassifier for experimenting with the built in parameters i.e. n_estimators, max_depth, and random_state. We tested the effect of the parameters by only changing the one that is currently being tested, and controlling the effect of other values by keeping them as default. We used sklearn's GridSearchCV to achieve this since GridSearchCV automatically tests every combination of an input parameter set via cross validation, default=5. The Experiment result is represented in Fig 10.

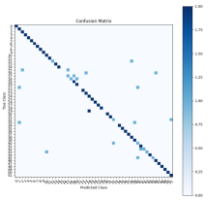
The n_estimators parameter is for choosing the number of the trees contained inside the forest. As its value increases, the training time and inferencing time also increases linearly. If the number of trees increases, the overall model becomes more stable and diverse, leading to a better generalization. Therefore we can expect to get better test accuracy if we increase the n_estimator parameter in trade-off for computation time.

The max_depth parameter is for choosing maximum depth that the tree can grow. If we increase this parameter, the tree can grow bigger and thus be able to learn more features for splitting data. We can see that as max_depth increases, accuracy rises at first but drops at some point. It

n_estimators	fit time (s)	score time (ms)	accuracy
10	0.4383	10.104	0.3487
20	0.8159	9.871	0.4400
30	0.9671	6.664	0.4593
40	0.9030	5.326	0.5025
50	1.1509	6.727	0.5098
60	1.5590	8.328	0.5266
70	1.7339	9.579	0.5339
80	1.8856	8.709	0.5460
90	2.2580	10.463	0.5316
100	2.4925	12.608	0.5892
110	2.7117	14.009	0.5555
120	2.9367	13.109	0.5820
130	3.1506	15.225	0.5797
140	3.8787	22.234	0.5580

max_depth	fit time (s)	score time (ms)	accuracy
1	0.5415	13.7619	0.1347
2	0.5041	9.7780	0.2188
3	0.8579	11.7547	0.2525
4	0.8102	8.2254	0.3367
5	0.9544	8.3095	0.3776
6	1.3519	10.4942	0.4135
7	1.2546	9.0476	0.4376
8	1.6084	12.7684	0.5025
9	1.4890	10.1818	0.4809
10	1.8401	10.4919	0.5316
11	1.9229	12.1740	0.5506
12	1.7715	9.9144	0.5363
13	2.0936	12.8891	0.5484
14	2.1550	11.6154	0.5868
15	2.1512	13.1091	0.5627
16	2.1195	10.0859	0.5436
17	2.3108	12.3195	0.5532
18	2.3360	13.2454	0.5458
19	2.3483	13.1969	0.5653

random_state	fit time (s)	score time (ms)	accuracy
0	2.3603	11.7325	0.5845
1	2.4653	12.7250	0.5770
2	2.4825	13.3955	0.5843
3	2.4718	11.5300	0.5772
4	2.5055	11.3313	0.5699
5	2.4681	11.8997	0.5434
6	2.5607	14.2937	0.5796
7	2.3089	11.0456	0.5723
8	2.4161	11.0782	0.5747
9	2.4722	11.6253	0.5794
10	2.4838	12.0446	0.5796
11	2.4632	12.9890	0.5698
12	2.4659	13.1502	0.6156
13	2.4810	11.4102	0.5844
14	2.3516	11.5074	0.5749



-	Accuracy
Train	0.592
Test	0.827

Figure 9 Random Foreset parameter test results and confusion matrix / Accuracy of the best estimator

is because as the tree's depth increases the tree becomes more overfitted to the training data, leading to lower test accuracy. The training time also increases as the size of the tree grows, which is why we have to choose the maximum depth more carefully.

The random_state parameter is for choosing the degree of randomness when building the tree. We can see that accuracy varies considerably by randomness. It is because choosing a random dimension for the split affects the tree's construction directly and causes the accuracy significantly. Among all trained classifiers we were able to choose the estimator that fits best to our data using best_estimator_ attribute. Figure 9 contains the confusion matrix and train/test accuracy of the best random forest classifier. You can see that test accuracy is higher than train's. This is a due to the characteristics of GridSearchCV, which calculates the average train score of each cross-validate process. Since train score from cross-validation increases over each process the average can be lower than the final trained model. Figure 10 represents the success and failure cases of the mentioned classifier.

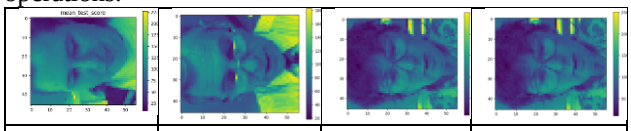
5.2 Effect of The Weak-Learners

In order to test the effect of weak-learners, i.e. axis-aligned and two-pixel test, we had to implement a custom random forest classifier(Implementation presented in Appendix C.1.). In this custom random forest classifier, we fixed the number of trees as 25 and maximum depth as 5. Also, to avoid the randomness effect we ran training steps 5 times and calculated average test accuracy and training time. The results are shown in Table 9.

Weak Learners	fit time(s)	Score time(ms)	accuracy
axis-aligned	65.2759	30.4355	0.3250
Two-pixel test	80.3541	41.3303	0.1942

Table 9 Comparison between weak-learners

Lots of questions arise from the above result. First question may be why the test accuracy is lower than the result we acquired from sklearn RandomForestClassifier. It is because the default number of estimators in sklearn's classifier is 100, but we only used 25 in our tests. Also, we cross validated the result 5 times when using the sklearn's built in function. Due to these reasons our current models were less generalized than the built-in model, but it wouldn't matter when testing the effect of different weak learners. The two-pixel test randomly chooses two features from the input and calculates the difference that makes the lowest impurity(=entropy) whereas axis-align only chooses one feature. Therefore the two-pixel test is capable of learning more complicated patterns from the image, so we first expected the two-pixel test to have better accuracy. But the results were the opposite. We carefully looked at the data's characteristic, and found that data's pixel values were very similar. Therefore using the difference of the pixels might not have contained much information. Also, we think when discriminating a person as a class, using a pixel value itself would be more useful than using pixel difference. (Lets say we are discriminating black man and white man, the pixel value of two people's face will differ will be greatly. But pixel difference value of their face will not be different as before). Due to these characteristic of the data, the axis-aligned's accuracy were better than two-pixel test. The total train time is shorter for axis-aligned, which seems intuitive since the two-pixel test runs more operations.



Input	Predict	Input	Predict

Figure 10 Random Forest prediction Failure/Success case

5.3 Comparison with Q1, Q3

Comparing with PCA and PCA-LDA model, We can see that random forest's accuracy is higher than PCA and lower than PCA-LDA ensemble. The reason that random forest's accuracy was higher than PCA is because random forests are able to fit into relatively complex model than PCA. Which leads to a better accuracy than PCA in our data. But using PCA-LDA ensemble tends to have better accuracy because LDA extracts information that are more useful for classification. Also, using various combination of model via ensemble increases the model's diversity and model's generalization. Which is why PCA-LDA ensemble had better accuracy than Random Forest.

Appendix

Appendix A
Multi-class SVM for Face Recognition

A.1. Effect of SVM parameters

In this section, we trained and tested Support Vector Machine using the given data with a different set of parameters to experiment how they affect the training time, inference time and test accuracy. We utilized the scikit-learn's SVC for experimenting with the built in parameters i.e. kernel_type, kernel_parameters, and C. The process is very similar to Section 5.1. We tested the effect of the parameters by only changing the one that is currently being tested, and controlling the effect of other values by keeping them as default. We again sklearn's GridSearchCV which helps us test every combination of an input parameter via cross validation. The Experiment result is represented in Table 10.

The kernel parameter is for specifying the kernel type to be used in the algorithm. In the result, we found that the linear kernel works best for our data. When using poly or rgf, the data will be mapped into a very high dimension creating a complex boundary. We think this can cause an overfitting to the training data, causing the test accuracy to be lower than linear, which creates a relatively simple boundary. Also our dataset may be linearly separable, therefore using the linear kernel leads to the highest test accuracy.

Parameter	values	fit time (s)	score time (ms)	accuracy
kernel	linear	0.1743	33.5211	0.5842
	poly	0.1642	33.0415	0.4904
	rgf	0.1911	64.6608	0.4376
	sigmoid	0.1829	34.1507	0.0048
gamma	scale	0.1954	62.0728	0.4376
	auto	0.1811	60.2570	0.0409
C	0.1	0.3169	162.7319	0.1562
	0.2	0.2388	91.9419	0.1562
	0.3	0.2365	83.7930	0.1610
	0.4	0.1950	64.7593	0.1706
	0.5	0.1928	65.2409	0.1778
	0.6	0.1951	66.4357	0.2163
	0.7	0.2002	66.1437	0.2956
	0.8	0.2016	67.5863	0.3654
	0.9	0.2080	74.9694	0.4087
	1.0	0.2054	67.3333	0.4376

Table 10 SVM parameters test results

The gamma parameter is for setting a maximum distance when creating a curve of a line. When it's set to scale, gamma is set to 1 / (n_features * X.var()) and 1 / n_features when it is set to auto. We can see that auto's accuracy is very low. In the Scikit Learn webpage, auto is marked as to be deprecated. Therefore, we are going to use 'scale' as gamma for A.2.

The C parameter is for choosing the rate for maximizing the margin against minimizing the separating error. Choosing the high C value makes separating error to be lower and may lead to overfitting in the training data. In the result, we can see that higher C value creates higher test accuracy. Therefore we can predict that in the default setting, the model was a little bit under fitted and choosing C =1.0 works best. Therefore we will use 1.0 as C value for A.2

A.2. Multi Class Extension of SVM

In general SVM is used for binary classification. But by one vs one / one vs rest technique we can expand it to multi classes. To compare the two techniques we implemented the 'one vs one' and 'one vs rest' on top of scikit-learn's SVC.

```
def train_svm_ovr(train_faces,train_labels):
    unique_labels = np.unique(train_labels)
    models = []
    for l in unique_labels:
        cur_labels = np.where(train_labels == l, 1, -1)
        model = SVC(kernel='linear',gamma='scale',class_weight='balanced')
        model.fit(train_faces,cur_labels)
        models.append(model)
    return models

def train_svm_ovo(train_faces,train_labels):
    unique_labels = np.unique(train_labels)
    label_combinations = combinations(unique_labels,2)
    models = []
    for l1,l2 in label_combinations:
        targets = np.where((train_labels == l1) | (train_labels == l2))
        cur_train_faces = train_faces[targets]
        cur_train_labels = train_labels[targets]
        cur_labels = np.where(cur_train_labels == l1, 1, -1)
        model = SVC(kernel='linear',gamma='scale',class_weight='balanced')
        model.fit(cur_train_faces,cur_labels)
        models.append((l1,l2,model))
    return models

def test_svm_ovr(models,test_faces):
    res = np.zeros((test_faces.shape[0], len(models)))
    for i, model in enumerate(models):
        res[:, i] = model.predict(test_faces)
    return np.argmax(res, axis=1)

def test_svm_ovo(models, test_faces):
    res = np.zeros((test_faces.shape[0], len(models)))
    for l1,l2, model in models:
        pred = model.predict(test_faces)
        res[np.where(pred == 1), l1] += 1
        res[np.where(pred == -1), l2] += 1
    return np.argmax(res, axis=1)
```

Above is the code for multi-class expansion. The SVC uses parameters that worked best in section A.1. Train and test results of above function is in Table 11.

Technique	fit time(s)	score time(ms)	accuracy
one-vs-one	2.7673	1814.7962	0.8173
one-vs-rest	3.6414	305.1703	0.7212

Table 11 Comparison between one-vs-one / one-vs-rest

The train time for one-vs-one is lower than one-vs-rest, which seems awkward because one-vs-one has to create more combinations of the classifiers. But we think that

each individual SVC in the one-vs-one model will train less data than one-vs-rest, because it only has to train data within two classes. Thanks to this effect, the training time for the one-vs-one model was able to be lower even though it had more classifiers to train. The test time however, takes longer because in this situation, both techniques have to gather prediction results from all of its classifiers. Since one-vs-one has nC2 classifiers and one-vs-rest has n classifiers, this difference yields a significant gap in the testing time. The one-vs-one model had better test accuracy than one-vs-rest model. We think since one-vs-one technique considers every combination of classes, it is more sensitive to inter-class's collinearity and yields more precise classification. Figure 13 contains the confusion matrix, success/failure images. Table 12 presents the example of the support vectors and margins from a randomly selected SVC from each techniques.

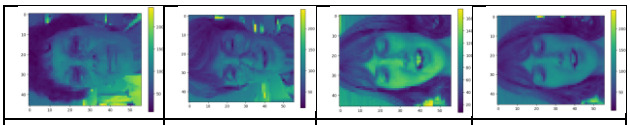


Figure 11 one-vs-one Failure/Success case

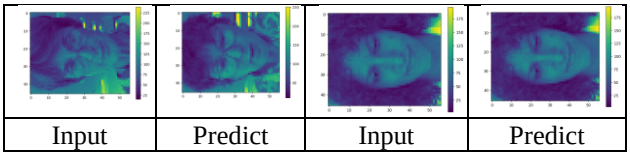


Figure 12 one-vs-rest Failure/Success case

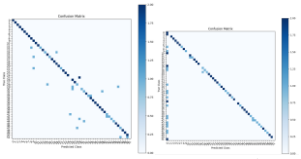


Figure 13 one-vs-one/one-vs-rest confusion matrix

Technique	class	margin	support_vectors
one-vs-one	(7,51)	1300.61	[[27. 29. 30. ... 80. 127. 181.] [215. 230. 237. ... 50. 45. 41.] [188. 186. 195. ... 60. 62. 58.] ... [116. 109. 115. ... 154. 178. 196.] [118. 108. 106. ... 66. 59. 92.] [104. 105. 100. ... 103. 107. 108.]]
one-vs-rest	18	480.21	[[169. 148. 126. ... 25. 23. 21.] [156. 182. 206. ... 86. 77. 82.] [85. 77. 56. ... 62. 62. 62.] ... [222. 231. 223. ... 143. 148. 150.] [87. 73. 59. ... 32. 31. 36.] [115. 115. 118. ... 58. 54. 51.]]

Table 12 Margin and Support Vectors of randomly selected SVC

A.3. Comparison to Others

Comparing prediction accuracy between SVM and PCA, SVM accuracy is higher than PCA. This is because PCA only cares about maximizing data variance and inter-class information might not be preserved. On the other hand, SVM is optimized for classifying the data. Therefore SVM having higher classification accuracy than PCA seems reasonable.

Prediction accuracy between SVM and random forest are similar. The classification accuracy being high or low can change due to random_state in random forest or regularization parameters of SVC. We think it is because SVM and random forests both can learn complex data distribution.

We can see that PCA-LDA has higher accuracy than SVM. Since PCA-LDA uses reduced dimension data, we believe PCA-LDA can be trained more generally without overfitting to the data. However, SVM has to learn directly from 2576 features. Which makes SVM fit into more complex function and leads it more prone to overfitting.

Appendix B
Visualization

B.1. PCA

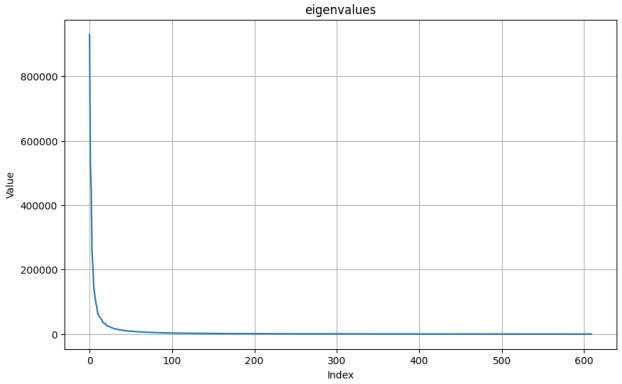


Figure 14 Eigenvalues of (1/N)AA^T

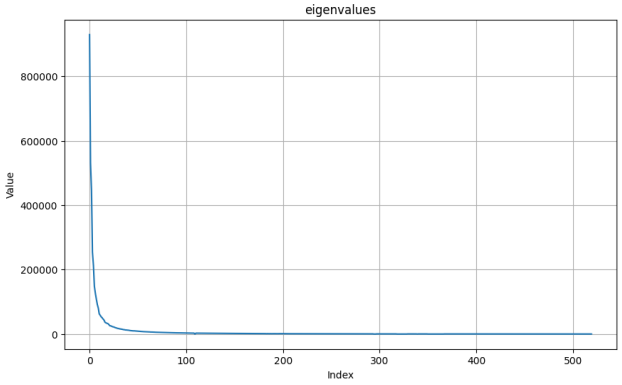


Figure 15 Eigenvalues of (1/N)A^TA

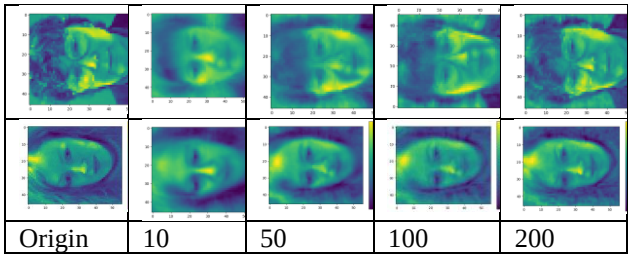


Figure 16 Image reconstruction by varying number of basis

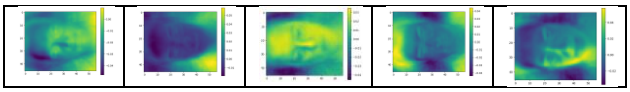


Figure 17 Eigenvector visualization of first 5

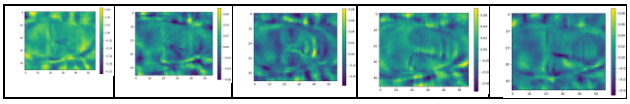


Figure 18 Eigenvector visualization of 51 to 55

Appendix C
Implementation Detail
C.1. Random Forest Implementation

```
import numpy as np
from collections import Counter
from itertools import combinations
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import random

class Node:
    '''
    Helper class which implements a single tree node.
    '''
    def __init__(self, feature=None, threshold=None, data_left=None, data_right=None,
gain=None, value=None, pixel1=None, pixel2=None):
        self.feature = feature
        self.threshold = threshold
        self.data_left = data_left
        self.data_right = data_right
        self.gain = gain
        self.value = value
        self.pixel1 = pixel1
        self.pixel2 = pixel2

class DecisionTree:
    '''
    Class which implements a decision tree classifier algorithm.
    '''
    def __init__(self, min_samples_split=2, max_depth=5, weak_learner='axis'):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.root = None
        self.pixel_diffs = None
        self.weak_learner = weak_learner

    @staticmethod
    def _entropy(s):
        '''
        Helper function, calculates entropy from an array of integer values.

        :param s: list
        :return: float, entropy value
        '''
        # Convert to integers to avoid runtime errors
        counts = np.bincount(np.array(s, dtype=np.int64))
        # Probabilities of each class label
        percentages = counts / len(s)

        # Caclulate entropy
        entropy = 0
        for pct in percentages:
            if pct > 0:
                entropy += pct * np.log2(pct)
        return -entropy

    def _information_gain(self, parent, left_child, right_child):
        '''
        Helper function, calculates information gain from a parent and two child nodes.

        :param parent: list, the parent node
        :param left_child: list, left child of a parent
        :param right_child: list, right child of a parent
        :return: float, information gain
        '''
        num_left = len(left_child) / len(parent)
        num_right = len(right_child) / len(parent)

        # One-liner which implements the previously discussed formula
        return self._entropy(parent) - (num_left * self._entropy(left_child) + num_right *
self._entropy(right_child))


    def _best_split(self, X, y):
        '''
        Helper function, calculates the best split for given features and target

        :param X: np.array, features
        :param y: np.array or list, target
        :return: dict
        '''
        best_split = {}
        best_info_gain = -1
        n_rows, n_cols = X.shape

        # For every dataset feature
        f_idx = random.randint(0,n_cols-1)



        X_curr = X[:, f_idx]
        # For every unique value of that feature

        #threshold = np.mean(X_curr)

        for threshold in np.unique(X_curr):
            # Construct a dataset and split it to the left and right parts
            # Left part includes records lower or equal to the threshold
            # Right part includes records higher than the threshold
            df = np.concatenate((X, y.reshape(1, -1).T), axis=1)
            df_left = np.array([row for row in df if row[f_idx] <= threshold])
            df_right = np.array([row for row in df if row[f_idx] > threshold])

            # Do the calculation only if there's data in both subsets
            if len(df_left) > 0 and len(df_right) > 0:
                # Obtain the value of the target variable for subsets
                y = df[:, -1]
                y_left = df_left[:, -1]
                y_right = df_right[:, -1]

                # Caclulate the information gain and save the split parameters
                # if the current split if better then the previous best
                gain = self._information_gain(y, y_left, y_right)
                if gain > best_info_gain:
                    best_split = {
                        'feature': f_idx,
                        'threshold': threshold,
                        'df_left': df_left,
                        'df_right': df_right,
                        'gain': gain
                    }
                    best_info_gain = gain
        return best_split

    # Two-pixel Test
    def _best_split_t(self, X, y):
        best_split = {}
        best_info_gain = -1
        n_rows, n_cols = X.shape
        f1_idx = random.randint(0,n_cols-1)
        f2_idx = random.randint(0,n_cols-1)
        while f2_idx == f1_idx:
            f2_idx = random.randint(0,n_cols-1)

        # For every dataset feature
        X_f1 = X[:, f1_idx]
        X_f2 = X[:, f2_idx]

        # Calculate the pixel differences
```

```
    pixel_diffs = X_f1 - X_f2

    # For every unique pixel difference value
    #pixel_diff = np.mean(pixel_diffs)
    # Construct a dataset and split it to the left and right parts
    for pixel_diff in np.unique(pixel_diffs):
      df = np.column_stack((X, y))
      df_left = df[pixel_diffs <= pixel_diff]
      df_right = df[pixel_diffs > pixel_diff]

      # Do the calculation only if there's data in both subsets
      if len(df_left) > 0 and len(df_right) > 0:
          y = df[:, -1]
          y_left = df_left[:, -1]
          y_right = df_right[:, -1]

          gain = self._information_gain(y, y_left, y_right)
          if gain > best_info_gain:
              best_split = {
                  'pixel1': f1_idx,
                  'pixel2': f2_idx,
                  'threshold': pixel_diff,
                  'pixel_diff': pixel_diff,
                  'df_left': df_left,
                  'df_right': df_right,
                  'gain': gain
              }
              best_info_gain = gain

    return best_split

def _build(self, X, y, depth=0):
    '''
    Helper recursive function, used to build a decision tree from the input data.

    :param X: np.array, features
    :param y: np.array or list, target
    :param depth: current depth of a tree, used as a stopping criteria
    :return: Node
    '''
    n_rows, n_cols = X.shape

    # Check to see if a node should be leaf node
    if n_rows >= self.min_samples_split and depth <= self.max_depth:
        # Get the best split
        if self.weak_learner == 'axis':
          best = self._best_split(X, y)
        else:
          best = self._best_split_t(X, y)
        #print(죄 고)
        # If the split isn't pure
        if best['gain'] > 0:
            # Build a tree on the left
            left = self._build(
                X=best['df_left'][:, :-1],
                y=best['df_left'][:, -1],
                depth=depth + 1
            )
            right = self._build(
                X=best['df_right'][:, :-1],
                y=best['df_right'][:, -1],
                depth=depth + 1
            )
            return Node(
                feature=best.get('feature'),
                threshold=best['threshold'],
                pixel1=best.get('pixel1'),
                pixel2=best.get('pixel2'),
                data_left=left,
                data_right=right,
                gain=best['gain']
            )
    # Leaf node - value is the most common target value
    return Node(
        value=Counter(y).most_common(1)[0][0]
    )

def fit(self, X, y):
    '''
    Function used to train a decision tree classifier model.

    :param X: np.array, features
    :param y: np.array or list, target
    :return: None
    '''
    # Call a recursive function to build the tree
    self.root = self._build(X, y)

def _predict(self, x, tree):
    '''
    Helper recursive function, used to predict a single instance (tree traversal).

    :param x: single observation
    :param tree: built tree
    :return: float, predicted class
    '''
    # Leaf node
    if tree.value != None:
        return tree.value

    if self.weak_learner == 'axis':
      feature_value = x[tree.feature]
    else:
      x = x.astype(np.float64)
      feature_value = x[tree.pixel1] - x[tree.pixel2]

    # Go to the left
    if feature_value <= tree.threshold:
        return self._predict(x=x, tree=tree.data_left)

    # Go to the right
    if feature_value > tree.threshold:
        return self._predict(x=x, tree=tree.data_right)

def predict(self, X):
    '''
    Function used to classify new instances.

    :param X: np.array, features
    :return: np.array, predicted classes
    '''
    # Call the _predict() function for every observation
    return [self._predict(x, self.root) for x in X]



class RandomForest:
    '''
    A class that implements Random Forest algorithm from scratch.
    '''
    def __init__(self, num_trees=25, min_samples_split=2, max_depth=5, weak_learner='axis'):
        self.num_trees = num_trees
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        # Will store individually trained decision trees
        self.decision_trees = []
        self.weak_learner = weak_learner

    @staticmethod
    def _sample(X, y):
        '''
        Helper function used for boostrap sampling.

        :param X: np.array, features
        :param y: np.array, target
        :return: tuple (sample of features, sample of target)
        '''
        n_rows, n_cols = X.shape
        # Sample with replacement
        samples = np.random.choice(a=n_rows, size=n_rows, replace=True)
        return X[samples], y[samples]

    def fit(self, X, y):
        '''
        Trains a Random Forest classifier.

        :param X: np.array, features
        :param y: np.array, target
        :return: None
        '''
        # Reset
```

```
    if len(self.decision_trees) > 0:
        self.decision_trees = []

    # Build each tree of the forest
    num_built = 0
    while num_built < self.num_trees:
        try:
            clf = DecisionTree(
                min_samples_split=self.min_samples_split,
                max_depth=self.max_depth,
                weak_learner=self.weak_learner,
            )
            # Obtain data sample
            _X, _y = self._sample(X, y)
            # Train
            clf.fit(_X, _y)
            # Save the classifier
            self.decision_trees.append(clf)
            num_built += 1
        except Exception as e:
            continue

def predict(self, X):
    '''
    Predicts class labels for new data instances.

    :param X: np.array, new instances to predict
    :return:
    '''
    # Make predictions with every tree in the forest
    y = []
    for tree in self.decision_trees:
        y.append(tree.predict(X))

    # Reshape so we can find the most common value
    y = np.swapaxes(a=y, axis1=0, axis2=1)

    # Use majority voting for the final prediction
    predictions = []
    for preds in y:
        counter = Counter(preds)
        predictions.append(counter.most_common(1)[0][0])
    return predictions
```