Project Part 2
Pattern Recognition
ECE 759

Kudiyar Orazymbetov (`korazym@ncsu.edu`)
Nico Casale (`ncasale@ncsu.edu`)

April 19, 2018

Contents *(Note that the entries are links.)*

List of Figures

List of Tables

Listings

1   Introduction

In this report, we outline the process we undertook to implement two classification algorithms: linear discriminant analysis (LDA) and Decision Trees. We tested our generated classifiers on the Yale Extended Face Dataset B and MNIST [**?**] datasets. To complete this, we experimented with two feature generation methods: principle components analysis (PCA) and LDA itself (for use in decision trees.) These were primarily useful in training our decision trees, which require a thorough exploration of the feature space to grow. Note that throughout this document, we use the terms *'features'* and *'attributes'* interchangeably.

After presenting an explanation of our algorithms' implementations, we will discuss our test results and preliminary exploration of the hyperparameter space. The hyperparameters are adjustments that variously control the accuracy and computational complexity of our classifiers. A more thorough exploration and validation of these parameters will be presented in the next installment of the project.

Code for this project is included in our project submission. In addition, it's available on GitHub at https://github.com/n-casale/ece759-project. The Git repository includes a more thorough representation of the path our code took to its current state.

2   Feature Selection

2.1   Decision Tree and Extra-Tree Feature Generation

In working with the decision trees, we utilized dimensionality reduction techniques including linear discriminant analysis (LDA) and principle components analysis (PCA) for training and testing classical decision trees, which pass over all features to consider which decision nodes to generate. This improved the computational requirements of the algorithm, as features in lower dimensions require less processing to distinguish.

LDA reduces data with K classes in large dimensions (on the order of the number of pixels) to K dimensions which captures the most energy of the data. Implementing LDA as dimensionality reduction, MNIST and Extended YaleB datasets are reduced to 10 and 38 variables respectively (for each data point.) We then worked with this transformed data as the input to our decision tree classifier for training and testing.

In a separate experiment, we considered the effects of utilizing the raw pixel data as features in *extra-trees* which are ensembles of decision trees trained on randomly chosen features and thresholds. When testing, the mode of the predicted classes is chosen as the global prediction of the ensemble of trees. This method is shown to be more accurate than our classical decision trees trained on PCA- and LDA-generated features in Section (4).

2.2   LDA vs PCA

LDA and PCA linearly transform the data to reduce its dimensionality. Dimensionality reduction is helpful in many algorithms because it eases computational requirements and improves the generalizability of the classifier. This latter benefit is conferred by the manner in which dimensionality reduction smoothes out the individual variations in a particular class. A good dimensionality reduction algorithm should maximize the 'distance' between classes in the transformed (reduced) space, while minimizing the variance, or spread of those classes in the transformed space.

The difference between the two techniques is that LDA is supervised whereas PCA is unsupervised. PCA finds orthogonal projections which capture the most variance of each feature, thereby expressing the data along *principle*

components that are the most distinct. We can visually exemplify the PCA technique as in the figure below.
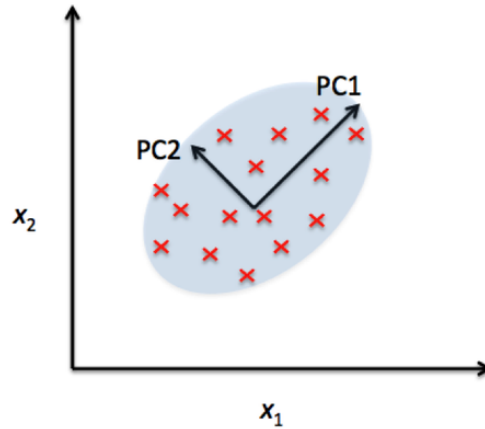

Figure 2.1: PCA implemented on 2-dimensional data [?].

In our PCA implementation, we rely on the *singular value decomposition* (SVD) to generate our features, conventionally described as *scores*. To perform the singular value decomposition and obtain our features, we must first pre-process the data. We arrange it so that the rows of our data X are the individual images in the dataset, with each column corresponding to a pixel value. Then, we subtract the column-wise empirical mean from each column to render each column zero-mean. Then, we utilize MATLAB's built-in `svd(.)` to obtain the matrices $[U, S, V]$. Our features are represented by $U * S$. Taking the transpose of this matrix, and extracting the first n columns, where n is the number of features we seek to utilize, we have a matrix of features corresponding to each image in our dataset. This method was useful, although not as effective in training decision trees as LDA, which we now discuss.

LDA maximizes the class separability and can be represented visually as in the figure below.
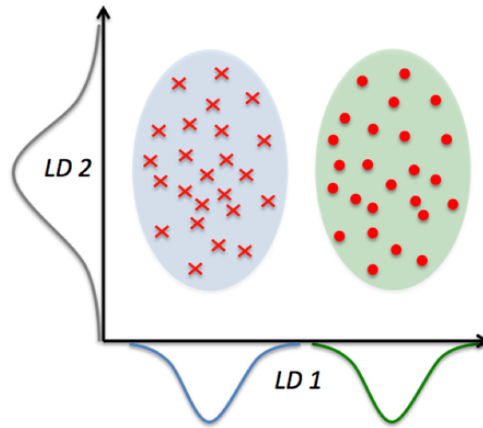
Figure 2.2: LDA visualization [?].

The underlying concept of LDA is taking the eigenvectors of $\dfrac{\Sigma_b}{\Sigma_w}$ where Σ_w is within-class scatter matrix and Σ_b is between-class scatter matrix. This division of matrices separates the classes away from one another. This is distinct from PCA, which is unsupervised, thereby only capturing the data in a minimum number of variables irrespective of class. Please see Section (3.3) for more information on LDA.

3   Algorithm Implementations

3.1   Decision Tree Algorithm

A binary decision tree is a hierarchical structure that takes input data at its root and propagates it to one of many leaves. Each *leaf* of the tree represents a class designation. To reach a leaf, the features of the data are utilized at *nodes* to make a binary decision: to proceed down the left or right *branch* of the tree? To answer this question, the node also carries a *threshold* that the feature value of the test data is compared against. If the test feature is less than the threshold, we proceed down the left branch. Otherwise, the right. An illustration of a simple decision tree is pictured below.

Figure 3.1: An example decision tree.
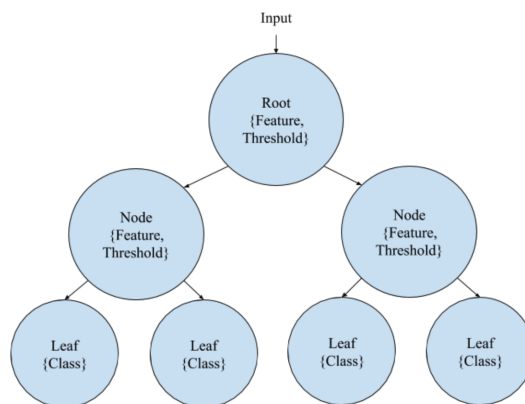
This decision tree structure needs to be generated before it can be used with test data. To train a decision tree that appropriately classifies our test data according to the features we generated, we employ a recursive function. The function signature is

$$\text{tree} = \texttt{trainDecisionTree(set)}$$

Where `set` is the training set, which is a MATLAB structure that contains the class labels and generated features. `tree` is the returned structure that can be used during testing. It is essentially a nested structure that contains two types of elements: nodes and leaves. At each node of the tree, a feature and threshold are specified. If a test sample's value at that particular feature is less than the threshold, the sample is passed down the left branch of the node. Similarly, if the sample's feature is greater than the threshold, it goes through the right branch. This is repeated until we reach a leaf node, which specifies a class membership.

The decision tree training algorithm has a few major steps, and proceeds by evaluating a metric called *information gain* at various configurations. For now, suffice it to say that information gain is a scalar that represents the improvement in prediction as we narrow down the set (by growing the tree) to find appropriate leaves. Our implementation most closely follows that of the ID3 and C4.5 algorithms developed by Ross Quinlan in the late 80s and early 90s [?] [?].

In general, the decision tree training algorithm is described qualitatively as follows:

1. Check stopping conditions, which generate leaves.
   - If there are no more features to split on, return a leaf with the class mode of the set.
   - The set is smaller than `minLeaf`, which is a tuning parameter that is meant to reduce overfitting of the training data. If this condition is met, return a leaf with the class mode of the set.

- If all samples in the set belong to the same class, return a leaf with the class.

- If no feature yields an improvement to the information gain (discussed below), then return a leaf with the class mode of the set. Note that this condition is only evaluated after step 2.

2. Iterate over each feature. Sort the set along the current feature. We utilize a threshold that splits the set between adjacent feature values such that the two subsets are composed of training samples whose features are less than and greater than the threshold respectively. Because the information gain across thresholds is convex on the whole (see Fig. 3.2), we use a line search that approximates the highest information gain for each threshold. This serves to reduce the computational complexity of our training algorithm.

   Let `attributeBest` and `indBest` be the feature and index that yield the highest information gain. Since the set is sorted, we can simply split the set at the index given by `indBest` for the recursion.

3. Recur over the two subsets given by `indBest` to find the next attribute that yields the highest information gain. Note that we exclude `attributeBest` in the recursion of `trainDecisionTree(.)` so that the same feature isn't chosen in the subset. In this way, the decision tree is grown so that it makes the most improvements to information gain at the nodes which are closest to the root.

The algorithm is also reproduced in pseudocode below.

Algorithm 3.1: trainDecisionTree

Data: *set* of training samples with class labels (*set*(1)) and attributes (features) (*set*(2)).
minLeaf, an integer specifying the minimum number of elements in *set* required to make a splitting node.
Result: *tree*, a structure containing nodes and leaves.

1 **begin**
2   check for base cases:
3   **if** $set(2) = \emptyset$ **then**
4   no more attributes (features) to split on.
5   **return** *leaf* with mode of *set*(1) (class labels)
6   **if** $length(set(1)) < minLeaf$ **then**
7   **return** *leaf* with mode of *set*(1) (class labels)
8     $thisSetEntropy \longleftarrow getEntropy(set)$
9   **if not** *thisSetEntropy* **then**
10   all samples are in the same class.
11   **return** *leaf* with first element of *set*(1)
12   instantiate tracking variables:
13     $attributeBest, thresholdBest, infoGainBest, indexBest \longleftarrow 0$
14   **for** $attribute \in range(\# \text{ of features left in set})$ **do**
15   sort *set* along *attribute*.
16   perform line search heuristic to approximate max information gain by splitting *set* at various indices.
17        $thisInfoGain \longleftarrow lineSearch(set)$
18   **if** $thisInfoGain > infoGainBest$ **then**
19           $attributeBest \longleftarrow attribute$
20           $infoGainBest \longleftarrow thisInfoGain$
21           $indexBest \longleftarrow$ index given by line search
22           $thresholdBest \longleftarrow$ attribute value midway between $indexBest$ and $indexBest + 1$
23   **if not** *infoGainBest* **or not** *attributeBest* **then**
24   no attribute provides an information gain.
25   **return** *leaf* with mode of *set*(1) (class labels)
26   re-sort *set* along *attributeBest*.
27     $subsets \longleftarrow getSubsets(set, attributeBest, indexBest)$
28     $subtree1 \longleftarrow trainDecisionTree(subsets(1), minLeaf)$
29     $subtree2 \longleftarrow trainDecisionTree(subsets(2), minLeaf)$
30   **return** *node* with $attributeBest$, $thresholdBest$, $subtree1$, and $subtree2$

Once we have trained the decision tree, we can begin to test it by passing the features generated from the test set through the tree. As the features are utilized in propagating through the tree, we eventually reach a leaf node and assign its associated class to the test vector. Testing results can be found in Section (4). The testing algorithm is reproduced in psuedocode below.

Algorithm 3.2: testDecisionTree

Data: *set* of test samples with class labels (*set*(1)) and attributes (features) (*set*(2)).
tree, the struct of structs that represents the trained decision tree.
Result: *set*, the input structure modified to include predicted class membership.

1 **begin**
2   **for each** *sample* ∈ *set* **do**
3   *treeWalked* ⟵ *tree*
4   *classified* ⟵ false
5   *attributes* ⟵ features corresponding to this sample
6   **while not** *classified* **do**
7   **if** *treeWalked* **is a** *node* **then**
8   *thisAttribute* ⟵ attribute given by *treeWalked*
9   *thisAttributeValue* ⟵ value of the sample's feature at *thisAttribute*
10   *thisThreshold* ⟵ threshold given by *treeWalked*
11   **remove** *thisAttribute* from the sample's feature vector.
12   **compare** *thisAttributeValue* to *thisThreshold*:
13   **if** *thisAttributeValue* < *thisThreshold* **then**
14   choose left branch.
15   *treeWalked* ⟵ left branch given by current *treeWalked*.
16   **else**
17   choose right branch.
18   *treeWalked* ⟵ right branch given by current *treeWalked*.
19   **else**
20   *treeWalked* is a leaf.
21   append predicted label to *set*(2) at this sample.
22   *classified* ⟵ true

With these two algorithms in place, we can train and test decision trees with varying depth and accuracy. Next, we discuss an adjacent concept that is integral to the decision tree algorithm.

3.1.1   Entropy and Information Gain

When growing the decision tree, the training algorithm utilizes a metric called *information gain* to optimize the predictive power of the tree. Before we can define information gain, we must first understand *entropy*. Entropy is an information theoretic concept that represents the amount of uncertainty in a given set of data. It is defined as

$$\text{H}(X) = -\sum_{i=1}^{n} \text{P}(x_i) \log_2 \text{P}(x_i) \tag{3.1}$$

Where $\text{P}(x_i)$ is the proportion of the number of elements with class x_i to the number of elements in the set X, and there are n classes in the set. Note that if all samples belong to class i, $\text{P}(x_i) = 1$ and $\log_2 \text{P}(x_i) = 0$, so the entropy is zero.

In calculating the information gain of splitting the set into two subsets, we utilize the entropy of the parent set and

subtract from it a weighted entropy of each subset [**?**]. This is better expressed by the following equation:

$$\text{IG}(X) = \text{H}(X) - \sum_{i=1}^{2} \frac{|S_i|}{|X|} \text{H}(S_i) \tag{3.2}$$

Where $|\cdot|$ is the cardinality, or number of elements in the set, and S_i are the two subsets composed of the elements of X partitioned across a given threshold for a given feature. The figure below is a plot of the information gain across splitting indices. So, after partitioning the set across a single attribute, (all elements of S_1 are less than the threshold, and all elements in S_2 are greater), we have a scalar value of information gain to decide which splitting threshold and feature would most improve the predictive ability of the decision tree. The plot below represents just one feature, but this calculation is required across all currently available features in the set. This quickly becomes computationally taxing for a large dataset, so we utilize a line search heuristic to improve the computational time without sacrificing a significant amount of accuracy.

The line search starts in the middle of the set and splits it, computing an information gain (IG). Then, we compare this middle IG to the IGs obtained by splitting the set at the 25% and 75% indices. If either of these reveals a larger IG, we set it to the new 'middle' index and consider the IGs which emerge from splitting the set half-way between the old 'middle' and half-way between the old 'left' or 'right' (depending on which one yielded a larger IG.) This allows us to approximate the maximum value of the information gain for a given feature without a brute-force technique of considering each threshold. We ensured that the line search incorporates a variety of locations to coax it to approaching the global maximum, rather than getting caught in the minor variations between adjacent indices. Please consult `code/decisionTree/trainDecisionTree.m` for further details.
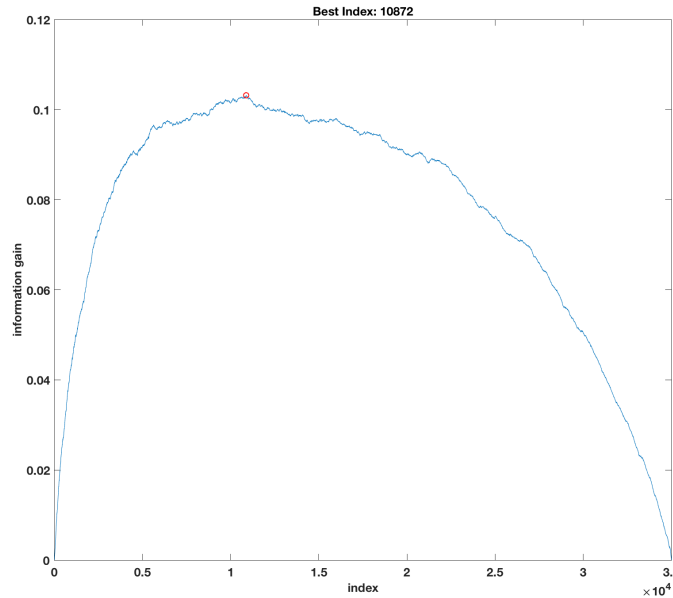

Figure 3.2: Information Gain across all possible thresholds.

3.2   Extra-Trees

The relative inadequacy of decision trees in classifying the samples of MNIST and Yale B led us to explore other tree-like classifiers. The first alternative that we considered is called *extra-trees*, which are ensembles of binary decision trees that are grown in a stochastic manner. In testing, these ensembles *vote* on the class by propagating the test sample down each tree until a leaf is found. The mode of all the votes is assigned to the global class prediction. See Section (4) for the result of this experiment.

Training extra-trees requires less computational and theoretical effort than classical decision trees. The reduced steps are certainly tangential to a basic decision tree, but incorporate less rigour. By utilizing the ensemble of trees, we can overcome the inaccuracies of a single random decision tree and make a strong prediction.

The steps to train an extra-tree are as follows:

1. Check stopping conditions, which generate leaves.
   - If there are no more features to split on, return a leaf with the class mode of the set.
   - The set is smaller than `minLeaf`, which is a tuning parameter that is meant to reduce overfitting of the training data. If this condition is met, return a leaf with the class mode of the set.
   - If all samples in the set belong to the same class, return a leaf with the class.

2. Choose a random feature. In extra-trees, we don't need to generate features, we can simply use the raw pixels as features.

3. Find the mean and variance of this feature across all samples in the set. Generate a random value from a normal distribution with this mean and variance.

4. Recur these steps on the subsets obtained by splitting the parent set on the randomly chosen feature and threshold, where the first subset contains samples whose feature is less than the threshold, and the second subset contains those which are greater than the threshold.

For MNIST, we generated 100 extra-trees, trained on half of the dataset. In testing, the majority-vote of the trees was used to assign a predicted class to the test vectors. Likewise, for Yale B, we utilized half the dataset and 100 extra-trees.

3.3   Linear Discriminant Analysis (LDA)

Our classification success criterion is the extent to which the error rate is minimized. For LDA, we employ the Bayes Rule to classify our test points after training the classifier. We assign the test point to the class with the highest conditional probability (*i.e.* $P(w_i|x)$, where w_i is the class and x the test vector). In practice, it is not feasible to get conditional probability for a given point unless we have a large amount of data. So we estimate the distribution and calculate the probabilities from there.

LDA relies on the assumption of a normal distribution for each class. Linear discriminant analysis generally achieves good performance in the tasks of face and object recognition, even though the assumptions of common covariance matrix among groups and normality are often violated [?].

Since the MNIST and Yale datasets are high-dimensional, we cannot check the normality of individual pixels. Instead, we reduce the dimensions via orthogonal projection. several classes (10 for MNIST and 38 for Yale B.) The class separation in a direction $\boldsymbol{w}$ in this case is given by the ratio [?].

$$S = \frac{\boldsymbol{w}^T \Sigma_b \boldsymbol{w}}{\boldsymbol{w} \Sigma \boldsymbol{w}} \tag{3.3}$$

In the case of two classes, this reduces to the ratio of between-class variance and within-class variance.

3.3.1   Algorithm steps

Below, we describe the steps necessary to construct the LDA classifier. We begin by instantiating some variables with notation:
n is the number of classes.
N is the total number of training data samples.
N_i is the number of points in each class i.
$\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}$ are mean vectors for each class and global mean for the data.

The steps that we follow in our LDA algorithm are as follows.

1. We calculate within-class scatter matrix for each class

$$\Sigma_i = \frac{1}{N_i - 1} \sum_{\boldsymbol{x} \in D_i}^{n} (\boldsymbol{x} - \boldsymbol{\mu}_i)(\boldsymbol{x} - \boldsymbol{\mu}_i)^T \tag{3.4}$$

then sum them to obtain

$$\Sigma_W = \sum_{i=1}^{n} (N_i - 1)\Sigma_i \tag{3.5}$$

2. We then find the average within-class scatter matrix by calculating

$$\Sigma = \frac{\Sigma_W}{N} \tag{3.6}$$

3. We also calculate the between-class scatter matrix by

$$\Sigma_B = \sum_{i=1}^{n} \frac{N_i}{N} (\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^T \tag{3.7}$$

4. We need to find eigenvectors and eigenvalues of $\Sigma^{-1}\Sigma_b$.

5. We then sort the eigenvectors depending on the magnitude of eigenvalues.

6. The number of highest eigenvalues will be $c - 1$ which will be 9 and 37 respectively for MNIST and Extended Yale datasets.

7. Project our data onto the subspace(constructed by the eigenvectors of the highest eigenvalues).

8. Since we have reduced dimensions of our data, we can easily apply a discriminant function for a test vector to see which class it belongs

9. The discriminant function for each class is

$$f_i(x_k) = \mu_i w_a^{-1} x_k^T - \frac{1}{2}\mu_i w_a^{-1} \mu_i^T + ln(P_i) \tag{3.8}$$

where P_i is the probability of each class.

10. Then we select the class with highest discriminant function evaluation.

3.3.2   Helpful Functions

In the above pseudo code, we used our own supplementary functions. After steps 1-7, there is a function `classify_comparison`, which applies the discriminant function in order to classify the test data. We also tried to calculate if the algorithm is effective in just using the distances from the centers of each class using `classify_from_centroid`. It did not work very well on MNIST dataset, but it does perform well with the Extended Yale B dataset.

3.3.3   Derivation of discriminant function

We classify the data point as being in class $\boldsymbol{i}$ if

$P(\boldsymbol{x}|i)P(i) > P(\boldsymbol{x}|j)P(j), \ \forall j \neq i.$

Assuming all covariances are equal, *i.e.* $\Sigma = \Sigma_i = \Sigma_j$, and that they follow a Multivariate Normal distribution, $P(\boldsymbol{x}|i) = \frac{1}{(2\pi)^{n/2}|\Sigma_i|^{1/2}} exp(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_i)^T \Sigma_i^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_i))$, the above condition becomes

$$\frac{P(i)}{(2\pi)^{n/2}|\Sigma_i|^{1/2}} exp(-\tfrac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_i)^T \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_i)) > \frac{P(j)}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp(-\tfrac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_j)^T \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_j))$$

Taking the logarithm of both sides,

$ln(|\Sigma|) - 2ln(P(i)) + (\boldsymbol{x} - \boldsymbol{\mu}_i)^T \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_i) < ln(|\Sigma|) - 2ln(P(j)) + (\boldsymbol{x} - \boldsymbol{\mu}_j)^T \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_j)$

After some algebraic manipulation, we have

$ln(P(i)) + \boldsymbol{\mu}_i \Sigma^{-1} \boldsymbol{x}^T - \tfrac{1}{2}\boldsymbol{\mu}_i \Sigma^{-1} \boldsymbol{\mu}_i^T > ln(P(j)) + \boldsymbol{\mu}_j \Sigma^{-1} \boldsymbol{x}^T - \tfrac{1}{2}\boldsymbol{\mu}_j \Sigma^{-1} \boldsymbol{\mu}_j^T$

Let $f_i = \boldsymbol{\mu}_i \Sigma^{-1} \boldsymbol{x}^T - \tfrac{1}{2}\boldsymbol{\mu}_i \Sigma^{-1} \boldsymbol{\mu}_i^T + ln(P(i))$. Then we can write

$f_i > f_j, \forall j \neq i$

as a measure of the class membership given the discriminant function.

4   Test Results

The table below illustrates the lowest classification errors we were able to achieve with our Decision Tree and LDA classifiers on the MNIST and Yale Extended B datasets. In LDA, we split the datasets in half so we can the other half for test. Using decision trees on MNIST, we utilized half of the dataset for training and half for testing. However, since the Yale B dataset is so small, we were only able to get lower error rates by using ∼95% of the set for training and ∼5% for testing (on decision trees.) For extra-trees, we split the two datasets in half to achieve our results.

Algorithm	MNIST	Yale B
LDA	13.5%	6.8%
Decision Tree	17.42%	57.89%
Extra-Trees	4.89% (100 extra-trees)	34.96% (100 extra-trees)

Table 4.1: Lowest classification errors achieved with LDA, Decision Trees, and Extra-Trees.

4.1   Cross Validation

In an effort to justify our choice of hyperparameters and examine the capability of our classifiers irrespective of training and test sets, we performed a 5-fold cross validation across the tuning parameters in each algorithm. For decision and extra trees, this consists of `minLeaf`. In k-fold cross validation, the classifier is trained k times on different subsets, ensuring that each sample of the dataset is used only once for validation. The figure below illustrates this process.
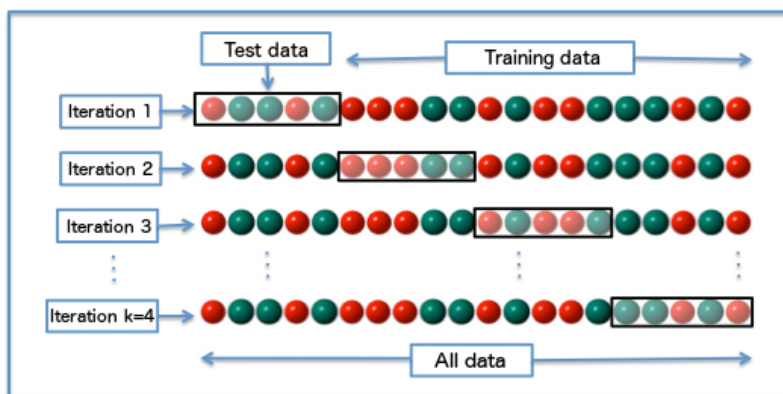
Figure 4.1: 4-fold cross validation on a dataset.

5 Analysis

5.1 Decision Trees

5.2 Extra Trees

5.3 LDA

6 Conclusion

In conclusion, we have shown that utilizing Linear Discriminant Analysis, Decision Trees, and Extra-Trees on the Extended Yale Dataset B and MNIST datasets yields a favorable result. In the next part of this project, we will improve our classification accuracy and partake in a more thorough exploration of the hyperparameter space to optimize and generalize our classifiers. Additionally, we will develop a stand-alone demonstration of our algorithms' process from training to testing.

7 Code Listings

Below are some of the primary scripts that execute the project. Please see the `code` folder for supporting functions and scripts.

Listing 1: Demonstration Code for Decision Tree on MNIST

```
1  %{
2
3  kudiyar orazymbetov
4  n casale
5
6  ECE 759 Project
7  18/03/16
8
```

```matlab
 9   this script orchestrates the training and testing of
10   the decision tree classifier
11
12   %}
13
14   function [errorRate] = demonstration_dt_MNIST()
15
16           addpath('utility', 'MNIST', 'MNIST/data', 'MNIST/loadMNIST', 'lda', ...
17                   'decisionTree');
18
19           fprintf('begin MNIST decision tree demonstration\n');
20
21           % hyper-parameters
22           N_tr = 35e3; % training samples
23           N_te = 35e3; % test samples
24
25           % for feature selection
26           numFeatures = 10;
27
28           % for decision tree
29           minLeaf = 1; % to prevent overfitting
30
31           % partition data
32           % MNIST contains 70k examples
33           [train, test] = loadMNIST(N_tr);
34
35           %% dimensionality reduction / feature generation
36           % via linear discriminant analysis (lda)
37           st = cputime;
38
39           [train, test] = lda_features(train, test, 0:numFeatures-1);
40
41           fprintf('Features Generated in %4.2f minutes\n', (cputime - st)/60);
42
43           %% train
44           st = cputime;
45
46           tree = trainDecisionTree({train{2:3}}, minLeaf);
47
48           fprintf('Trained in %4.2f minutes\n',(cputime - st)/60);
49
50           %% test
51           st = cputime;
52
53           test = testDecisionTree(test, tree);
54
55           fprintf('Tested in %4.2f minutes\n', (cputime - st)/60);
56
57           % Classification Error
58           errors = nnz(test{2}(:,1) ~= test{2}(:,2));
59           errorRate = (errors/N_te)*100;
60
61           fprintf('\nnumFeatures: %d, minLeaf: %d, error rate: %2.2f\n', ...
62                   numFeatures, minLeaf, errorRate);
63
64   end
```

Listing 2: Demonstration Code for Extra-Trees on Yale B

```matlab
%{

kudiyar orazymbetov
n casale

ECE 759 Project
18/03/16

this script orchestrates the training and testing of
the decision tree classifier

%}

function [errorRate] = demonstration_dt_MNIST()

        addpath('utility', 'MNIST', 'MNIST/data', 'MNIST/loadMNIST', 'lda', ...
                'decisionTree');

        fprintf('begin MNIST decision tree demonstration\n');

        % hyper-parameters
        N_tr = 35e3; % training samples
        N_te = 35e3; % test samples

        % for feature selection
        numFeatures = 10;

        % for decision tree
        minLeaf = 1; % to prevent overfitting

        % partition data
        % MNIST contains 70k examples
        [train, test] = loadMNIST(N_tr);

        %% dimensionality reduction / feature generation
        % via linear discriminant analysis (lda)
        st = cputime;

        [train, test] = lda_features(train, test, 0:numFeatures-1);

        fprintf('Features Generated in %4.2f minutes\n', (cputime - st)/60);

        %% train
        st = cputime;

        tree = trainDecisionTree({train{2:3}}, minLeaf);

        fprintf('Trained in %4.2f minutes\n',(cputime - st)/60);

        %% test
        st = cputime;

        test = testDecisionTree(test, tree);

        fprintf('Tested in %4.2f minutes\n', (cputime - st)/60);
```

```matlab
56
57          % Classification Error
58          errors = nnz(test{2}(:,1) ~= test{2}(:,2));
59          errorRate = (errors/N_te)*100;
60
61          fprintf('\nnumFeatures: %d, minLeaf: %d, error rate: %2.2f\n', ...
62                  numFeatures, minLeaf, errorRate);
63
64 end
```

Listing 3: Cross Validation Code for Decision Tree on MNIST

```matlab
 1 %{
 2
 3 kudiyar orazymbetov
 4 n casale
 5
 6 ECE 759 Project
 7 18/03/16
 8
 9 this script orchestrates the cross validation of
10 the decision tree classifier
11 across all hyperparameters,
12 taking performance and time results along the way
13
14 features are generated using lda, as they yield the best performance
15
16 %}
17
18 clear; close all;
19 addpath('utility', 'MNIST', 'MNIST/data', 'MNIST/loadMNIST', 'lda', ...
20          'decisionTree');
21
22 fprintf('begin Cross Validation on MNIST decision trees\n');
23
24 % hyper-parameters
25 k = 5; % k-fold cross validation
26 % use k to partition data
27 N_te = 70e3/k;
28 N_tr = 70e3 - N_te;
29
30 % for lda
31 numFeatures = 10;
32
33 % for decision tree
34 minLeaves = 1:10:100;
35
36 % partition data
37 % MNIST contains 70k examples
38 [train, test] = loadMNIST(N_tr);
39
40 % merge sets, they're already randomly shuffled
41 all = {[train{1}, test{1}],[train{2}; test{2}]};
42
43 %% k-fold cross validation across minLeaf
44
45 trainTimes = zeros(minLeaves(end), k);
```

```
46  errorRates = zeros(minLeaves(end), k);
47
48  for minLeaf = minLeaves
49
50          % random indices
51          inds = randperm(length(all{2}));
52          all{1} = all{1}(:, inds);
53          all{2} = all{2}(inds);
54
55          for fold = 1:k
56
57                  % choose fold
58                  inds_bool = false(70e3,1);
59                  ind_start = (fold-1)*N_te + 1;
60                  inds_bool(ind_start:ind_start+N_te-1) = 1;
61
62                  test = {all{1}(:,inds_bool), all{2}(inds_bool)};
63                  train = {all{1}(:,~inds_bool), all{2}(~inds_bool)};
64
65                  % dimensionality reduction / feature generation
66                  [train, test] = lda_features(train, test, 0:numFeatures-1);
67
68                  % train
69                  st = cputime;
70                  tree = trainDecisionTree({train{2:3}}, minLeaf);
71                  thisTrainTime = (cputime - st)/60;
72                  fprintf('Trained in %4.2f minutes\n', thisTrainTime);
73                  trainTimes(minLeaf, fold) = thisTrainTime;
74
75                  % test
76                  test = testDecisionTree(test, tree);
77
78                  % Classification Error
79                  thisError = nnz(test{2}(:,1) ~= test{2}(:,2));
80                  thisErrorRate = (thisError/N_te)*100;
81                  errorRates(minLeaf, fold) = thisErrorRate;
82
83                  fprintf('minLeaf: %d, fold: %d, error rate: %2.2f\n', ...
84                          minLeaf, fold, thisErrorRate);
85
86          end
87
88  end
89
90  %% save, print results
91  %filename = sprintf('UV%2.0f.mat', errorRate);
92  %save(filename, 'U', 'V', '-v7.3');
```

Listing 4: Main Code for LDA on MNIST

```
1  %{
2
3  kudiyar orazymbetov
4  n casale
5
6  ECE 759 Project
7  18/03/16
```

```matlab
 8
 9  this script orchestrates the training and testing of
10  the linear discriminant analysis classifier
11
12  %}
13
14  clear; close all;
15  addpath('../utility', '../MNIST', '../MNIST/data', '../MNIST/loadMNIST', ...
16          '../lda');
17
18  seed = 152039828;
19  rng(seed); % for reproducibility
20
21  % define parameters
22  N_tr = 35e3; % training samples
23  N_te = 35e3; % test samples
24  k = 10; % number of classes
25  % partition data
26
27  %{
28          1/2 of the dataset should be for training
29          the other for testing
30
31          MNIST contains 70k examples
32  %}
33
34  [train, test] = loadMNIST(N_tr);
35  %% use PCA to see the results
36  % numFeatures = 20;
37  % [train, U, V] = pca_(train, numFeatures);
38
39  %% Construct scatter matrices and calculate within-class and between class
40  % covariance
41  mu = mean(train{1,1}, 2);
42  num_variables = size(train{1,1},1);
43  % Let's standardize the data;
44  %variance = var(train{1,1}, 0,2);
45  %train{1,1} = (train{1,1}-repmat(mu,1, 60000))./variance;
46
47  %
48  Si = zeros(num_variables); Sb = zeros(num_variables);
49  S_cov = zeros(num_variables);
50  for i = 0:k-1
51      ind = (train{1,2} == i);
52      N_i = sum(ind);
53      x = train{1,1}(:, ind);
54      mu_i = mean(x, 2);
55      S_cov = S_cov + cov(x');
56      Si = Si + (1/N_tr)*(x - (repmat(mu_i,1, N_i)))*(x - (repmat(mu_i,1, N_i)))
57          ';
57      Sb = Sb + (N_i/N_tr)*(mu_i - mu)*(mu_i - mu)'; % (1/k)
58  end
59
60  % We apply singular value decomposition in order to find eigenvalues and
61  % eigenvectors
62  [U D V] = svd(pinv(Si)*Sb); % lets try S_cov/k instead of Si; but it is the
```

17

```matlab
      same result
63  a = [];
64  for i = 1:(k)
65      a = [a D(i,i)];
66  end
67
68  % from here we can see that we only have 9 highest values as we expected
69
70  % We transform the training and testing data to a subspace
71  transf_matrix = U(:,1:(k-1));
72  transf_train = train{1,1}'* transf_matrix;
73  transf_test = test{1,1}'*transf_matrix;
74  % We find mean vector and covariance matrix for each class
75  mu_each_class = zeros(k-1, k);
76  cov_each_class = {};
77  sum_cov = zeros(k-1,k-1);
78  for i =0:k-1
79      ind = find(train{1,2} == i);
80      X = transf_train(ind, :);
81      mu_each_class(:, i+1) = mean(X, 1)';
82      cov_each_class{1, i+1} = cov(X);
83      sum_cov = sum_cov + cov(X);
84  end
85  % since we assume equal covariance in all classes, we take the average of
86  % covariance matrices
87  average_cov = sum_cov/k;
88  cov_equal_each_class = {average_cov average_cov average_cov average_cov
        average_cov average_cov average_cov average_cov average_cov average_cov};
89  % this part is just a test on how nearest neigbors work
90  % parfor n = 1:13
91  % % we apply Nearest neigbors in order to find which class it belongs
92  %     accuracy(n) = classifyNN(n,transf_test', transf_train', test{1,2}, train
        {1,2});
93  % end
94  [acc_test acc_train] = classify_comparison_same_cov(k,5,mu_each_class,
        average_cov, transf_test', test{1,2}, transf_train', train{1,2}); % 0.88
        and 0.89 resp using just knn
95  %[acc_test_comp acc_train_comp] = classify_comparison(k,5,mu_each_class,
        cov_equal_each_class, transf_test', test{1,2}, transf_train', train{1,2});
        % 0.88 and 0.89 resp using just knn
96
97  %% This part uses kNN in order to class after transformation
98  [acc_test_5 acc_train_5] = classifyNN(k,5,mu_each_class, cov_each_class,
        transf_test', test{1,2}, transf_train', train{1,2}); % 0.88 and 0.89 resp
        using just knn
99  [acc_test_5 acc_train_5] = classifyNN(k,5,mu_each_class, cov_equal_each_class,
        transf_test', test{1,2}, transf_train', train{1,2}); % 0.87 and 0.88 resp
        using just knn
100 [acc_test_5_p acc_train_5_p] = classifyNN_pure(5,transf_test', transf_train',
        test{1,2}, train{1,2});
101
102 % cov each class separately works better in each case
103 % we plot the results to see the best number of nearest neighbors
104 % [acc_test_5 acc_train_5] = classifyNN(5,transf_test', transf_train', test
        {1,2}, train{1,2}); % 0.86 and 0.85 resp using just knn
105 % this is my first attempt
```

```matlab
106
107  f = instantiateFig(1);
108  plot([1:13],accuracy*100, 'r.')
109  prettyPictureFig(f);
110  xlabel('Nearest neighbor number');
111  ylabel('Accuracy of test model');
112
113  print('../../images/NN after LDA', '-dpng');
114  % instead we can use Euclidean distance metric to evaluate the classes by
115  % calculating the distances from each class centroid
116  centroid = zeros(k, k-1);
117  for i = 0:k-1
118      ind = (train{1,2} == i);
119      N_i = sum(ind);
120      centroid(i+1, :) = mean(transf_train(ind,:), 1);
121  end
122
123  accuracy1_train = classify_from_centroid(transf_train', train{1,2}, centroid);
124  accuracy1 = classify_from_centroid(transf_test', test{1,2},centroid);
125  %%
126  t0 = cputime;
127  N_cross_val = 5;
128  CVO = cvpartition(train{1,2},'k',N_cross_val);
129  err = zeros(CVO.NumTestSets,1);
130  for j = 1:N_cross_val
131      ind = CVO.training(j);
132      train_cv = transf_train(ind,:); train_cv_label = train{1,2}(ind,:);
133      test_cv = transf_train(~ind,:); test_cv_label = train{1,2}(~ind);
134      mu_each_class = zeros(k-1, k);
135      sum_cov = zeros(k-1,k-1);
136      for i = 0:(k-1)
137          ind1 = find(train_cv_label == i);
138          X = train_cv(ind1, :);
139          mu_each_class(:, i+1) = mean(X, 1)';
140          sum_cov = sum_cov + cov(X);
141      end
142      E_cov = sum_cov/k;
143      [acc_cross_valid_test(j) acc_cross_valid_train(j)] =
144          classify_comparison_same_cov(k,5,mu_each_class, E_cov, test_cv',
145          test_cv_label, train_cv', train_cv_label);
144  end
145  mean_acc_test = mean(acc_cross_valid_test);
146  sd_test = sqrt(var(acc_cross_valid_test));
147  mean_acc_train = mean(acc_cross_valid_train);
148  sd_train = sqrt(var(acc_cross_valid_train));
149  fprintf('Tested in %4.2f minutes\n', (cputime - t0)/60);
150
151
152  for i =0:k-1
153      ind = find(train{1,2} == i);
154      X = transf_train(ind, :);
155      mu_each_class(:, i+1) = mean(X, 1)';
156      cov_each_class{1, i+1} = cov(X);
157      sum_cov = sum_cov + cov(X);
158  end
159  % since we assume equal covariance in all classes, we take the average of
```

```
160  % covariance matrices
161  average_cov = sum_cov/k;
162  %%
163  load('fisheriris');
164  CVO = cvpartition(species,'k',10);
165  err = zeros(CVO.NumTestSets,1);
166  for i = 1:CVO.NumTestSets
167      trIdx = CVO.training(i);
168      teIdx = CVO.test(i);
169      ytest = classify(meas(teIdx,:),meas(trIdx,:),...
170                  species(trIdx,:));
171      err(i) = sum(~strcmp(ytest,species(teIdx)));
172  end
173  cvErr = sum(err)/sum(CVO.TestSize);
174
175
176  %% this is classification through matlab discriminant classification
177  mdl = fitcdiscr(transf_train, train{1,2},'DiscrimType','linear'); % this one
         works since n>m in tansf_train
178  %mdl = fitcdiscr(train{1,1}', train{1,2},'DiscrimType','linear'); % error
         saying Predictor x1 has zero within-class variance.
179  pred = predict(mdl, transf_test);
180  count = 0;
181  for i = 1:size(transf_test,1)
182     if pred(i) ==  test{1,2}(i);
183         count = count + 1;
184     end
185  end
186  acc = count/size(transf_test,1) % 0.8639
187  % we get a similar result as our methodp
188  % Lets try the same thing in decision tree
189  mdl_tree = fitctree(transf_train, train{1,2});
190  pred = predict(mdl_tree, transf_test);
191  count = 0;
192  for i = 1:size(transf_test,1)
193     if pred(i) ==  test{1,2}(i);
194         count = count + 1;
195     end
196  end
197  acc_tree = count/size(transf_test,1)
198  % this result is from pca
199  mdl_tree_pca = fitctree(train{3}, train{2});
200  pred = predict(mdl_tree_pca, test{3});
201  count = 0;
202  for i = 1:size(test{2},1)
203     if pred(i) ==  test{2}(i);
204         count = count + 1;
205     end
206  end
207  acc_tree_pca = count/size(test{2},1)
208  % with pure data
209  mdl_tree_pca = fitctree(train{1}', train{2});
210  pred = predict(mdl_tree_pca, test{1}');
211  count = 0;
212  for i = 1:size(test{2},1)
213     if pred(i) ==  test{2}(i);
```

```matlab
214          count = count + 1;
215      end
216  end
217  acc_tree_pca = count/size(test{2},1)
```

Listing 5: Main Code for LDA on Yale B

```matlab
1  %{
2
3  kudiyar orazymbetov
4  n casale
5
6  ECE 759 Project
7  18/03/16
8
9  this script orchestrates the training and testing of
10 the linear discriminant analysis classifier
11
12 %}
13
14 clear; close all;
15 addpath('../utility');
16 addpath('../YaleB', '../YaleB/data', '../YaleB/50Train', '../decision tree');
17 %% before
18 % [faces, labels] = loadYaleB();
19 % labels = labels - ones(size(labels,1));
20 % seed = 152039828;
21 % rng(seed); % for reproducibility
22 % load('2.mat'); % we load the indices to train and test sets
23 % train = {faces(trainIdx, :)', labels(trainIdx)};
24 % test = {faces(testIdx, :)', labels(testIdx)};
25 % k = 38;
26 % N_tr = size(trainIdx,1);
27 %%
28 [train, test] = loadYaleB();
29 train{1,2} = train{1,2} - 1;
30 test{1,2} = test{1,2} - 1;
31 k = 38;
32 N_tr = size(test{1,2},1);
33 %% use PCA to see the results
34 % numFeatures = 50;
35 % [train, U, V] = pca_(train, numFeatures);
36 % train{1,1} = train{1,3};
37 % [test, U, V] = pca_(test, numFeatures);
38 % test{1,1} = test{1,3};
39 %% Construct scatter matrices and calculate within-class and between class
40 % covariance
41 mu = mean(train{1,1}, 2);
42 num_variables = size(train{1,1},1);
43 % Let's standardize the data;
44 %variance = var(train{1,1}, 0,2);
45 %train{1,1} = (train{1,1}-repmat(mu,1, 1900))./variance;
46 %
47 Si = zeros(num_variables); Sb = zeros(num_variables);
48 for i = 0:k-1
49     ind = (train{1,2} == i);
50     N_i = sum(ind);
```

21

```matlab
51        x = train{1,1}(:, ind);
52        mu_i = mean(x, 2);
53        Si = Si + (1/N_tr)*(x - (repmat(mu_i,1, N_i)))*(x - (repmat(mu_i,1, N_i)))
              ';
54        Sb = Sb + (N_i/N_tr)*(mu_i - mu)*(mu_i - mu)'; % (1/k)
55 end
56
57 % We apply singular value decomposition in order to find eigenvalues and
58 % eigenvectors
59 [U D V] = svd(pinv(Si)*Sb);
60 a = [];
61 % for i = 1:(k)
62 %     a = [a D(i,i)];
63 % end
64 % from here we can see that we only have 9 highest values as we expected
65 %% this way is done using pca
66 % We transform the training and testing data to a subspace
67 % transf_train = train{1,1}'* U(:,1:numFeatures);
68 % transf_test = test{1,1}'*U(:, 1:numFeatures);
69 % % calculting the multivariate parameters
70 % mu_each_class = zeros(numFeatures, k);
71 % cov_each_class = {};
72 % sum_cov = zeros(numFeatures,numFeatures);
73 % for i =0:k-1
74 %     ind = find(train{1,2} == i);
75 %     X = train{1,1}';
76 %     X = X(ind, :);
77 %     mu_each_class(:, i+1) = mean(X, 1)';
78 %     cov_each_class{1, i+1} = cov(X);
79 %     sum_cov = sum_cov + cov(X);
80 % end
81 % average_cov = sum_cov/k;
82 %
83 % cov_equal_each_class = {average_cov average_cov average_cov average_cov
      average_cov average_cov average_cov average_cov average_cov average_cov
      average_cov average_cov average_cov average_cov average_cov average_cov
      average_cov average_cov average_cov average_cov average_cov average_cov
      average_cov average_cov average_cov average_cov average_cov average_cov
      average_cov average_cov average_cov average_cov average_cov average_cov
      average_cov average_cov average_cov average_cov};
84 % % to see how our model works
85 % [acc_test_comp acc_train_comp] = classify_comparison(k,5,mu_each_class,
      cov_equal_each_class, test{1,1}, test{1,2}, train{1,1}, train{1,2}); % 0.88
       and 0.89 resp using just knn
86
87 %% We transform the training and testing data to a subspace
88 transf_train = train{1,1}'* U(:,1:(k-1));
89 transf_test = test{1,1}'*U(:, 1:(k-1));
90 % calculting the multivariate parameters
91 mu_each_class = zeros(k-1, k);
92 cov_each_class = {};
93 sum_cov = zeros(k-1,k-1);
94 for i =0:k-1
95     ind = find(train{1,2} == i);
96     X = transf_train(ind, :);
97     mu_each_class(:, i+1) = mean(X, 1)';
```

```matlab
98        cov_each_class{1, i+1} = cov(X);
99        sum_cov = sum_cov + cov(X);
100  end
101  average_cov = sum_cov/k;
102  cov_equal_each_class = {average_cov average_cov average_cov average_cov
        average_cov average_cov average_cov average_cov average_cov average_cov
        average_cov average_cov average_cov average_cov average_cov average_cov
        average_cov average_cov average_cov average_cov average_cov average_cov
        average_cov average_cov average_cov average_cov average_cov average_cov
        average_cov average_cov average_cov average_cov average_cov average_cov
        average_cov average_cov average_cov average_cov};
103  % to see how our model works
104  [acc_test_comp acc_train_comp] = classify_comparison_same_cov(k,5,mu_each_class
        , average_cov, transf_test', test{1,2}, transf_train', train{1,2}); % 0.88
        and 0.89 resp using just knn
105  %[acc_test_comp acc_train_comp] = classify_comparison(k,5,mu_each_class,
        cov_equal_each_class, transf_test', test{1,2}, transf_train', train{1,2});
        % 0.88 and 0.89 resp using just knn
106  %%
107  t0 = cputime;
108  N_cross_val = 5;
109  CVO = cvpartition(train{1,2},'k',N_cross_val);
110  err = zeros(CVO.NumTestSets,1);
111  for j = 1:N_cross_val
112      ind = CVO.training(j);
113      train_cv = transf_train(ind,:); train_cv_label = train{1,2}(ind,:);
114      test_cv = transf_train(~ind,:); test_cv_label = train{1,2}(~ind);
115      mu_each_class = zeros(k-1, k);
116      sum_cov = zeros(k-1,k-1);
117      for i = 0:(k-1)
118          ind1 = find(train_cv_label == i);
119          X = train_cv(ind1, :);
120          mu_each_class(:, i+1) = mean(X, 1)';
121          sum_cov = sum_cov + cov(X);
122      end
123      E_cov = sum_cov/k;
124      [acc_cross_valid_test(j) acc_cross_valid_train(j)] =
            classify_comparison_same_cov(k,5,mu_each_class, E_cov, test_cv',
            test_cv_label, train_cv', train_cv_label);
125  end
126  mean_acc_test = mean(acc_cross_valid_test);
127  sd_test = sqrt(var(acc_cross_valid_test));
128  mean_acc_train = mean(acc_cross_valid_train);
129  sd_train = sqrt(var(acc_cross_valid_train));
130  fprintf('Tested in %4.2f minutes\n', (cputime - t0)/60);
131  %% this part is for testing with kNN
132  [acc_test_5 acc_train_5] = classifyNN(k,5,mu_each_class, cov_equal_each_class,
        transf_test', test{1,2}, transf_train', train{1,2}); % 0.88 and 0.89 resp
        using just knn
133  [acc_test_5_p acc_train_5_p] = classifyNN_pure(5,transf_test', transf_train',
        test{1,2}, train{1,2});
134
135  % let's now change the n's to see how it works
136  parfor n = 1:13
137  % we apply Nearest neigbors in order to find which class it belongs
138      accuracy(n) = classifyNN(n,transf_test', transf_train', test{1,2}, train
```

```
            {1,2});
139  end
140  % we plot the results to see the best number of nearest neighbors
141
142  f = instantiateFig(1);
143  plot([1:13],accuracy*100, 'r.')
144  prettyPictureFig(f);
145  xlabel('Nearest neighbor number');
146  ylabel('Accuracy of test model');
147
148  print('../../images/YaleBNNafterLDA', '-dpng');
149  % instead we can use Euclidean distance metr
150  ic to evaluate the classes by
151  %% calculating the distances from each class centroid
152  % for our surprise, it performs really well
153  centroid = zeros(k, k-1);
154  for i = 0:k-1
155      ind = (train{1,2} == i);
156      N_i = sum(ind);
157      centroid(i+1, :) = mean(transf_train(ind,:), 1);
158  end
159
160  accuracy1 = classify_from_centroid(transf_test', test{1,2},centroid);
161  accuracy1_train = classify_from_centroid(transf_train', train{1,2},centroid);
```