Project Part 2
Pattern Recognition
ECE 759

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

April 20, 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 [2] 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 exploration of the hyperparameter space. The hyperparameters are adjustments that variously control the accuracy and computational complexity of our classifiers. We utilized cross validation to verify our classifiers' stability and our optimal choice of hyperparameters.

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.

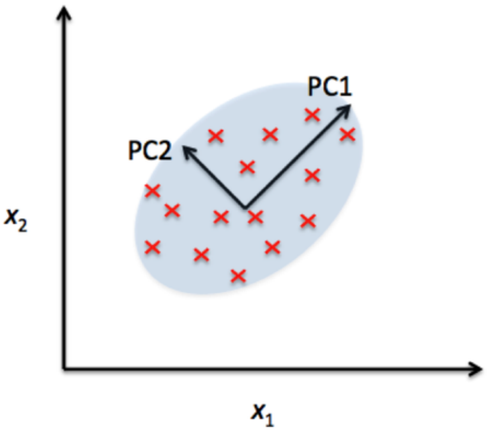


Figure 2.1: PCA implemented on 2-dimensional data [1].

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.

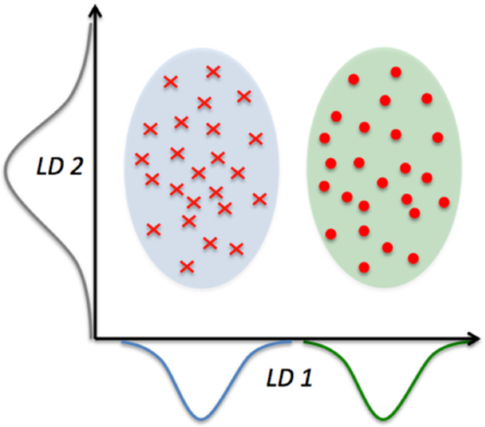


Figure 2.2: LDA visualization [1].

The underlying concept of LDA is taking the eigenvectors of $\frac{\Sigma_b}{\Sigma_w}$ 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.

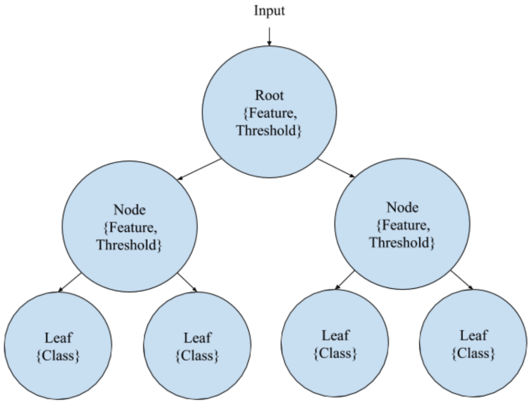

Input
Root {Feature, Threshold}
Node {Feature, Threshold}
Node {Feature, Threshold}
Leaf {Class}
Leaf {Class}
Leaf {Class}
Leaf {Class}

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

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

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 [3] [4].

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.

```
begin
    check for base cases:
    if $set(2) = \emptyset$ then
        no more attributes (features) to split on.
        return leaf with mode of $set(1)$ (class labels)
    if $length(set(1)) < minLeaf$ then
        return leaf with mode of $set(1)$ (class labels)
    $thisSetEntropy \longleftarrow getEntropy(set)$
    if not $thisSetEntropy$ then
        all samples are in the same class.
        return leaf with first element of $set(1)$
    instantiate tracking variables:
    $attributeBest, thresholdBest, infoGainBest, indexBest \longleftarrow 0$
    for $attribute \in range(\#\ of\ features\ left\ in\ set)$ do
        sort set along attribute.
        perform line search heuristic to approximate max information gain by splitting set at various indices.
        $thisInfoGain \longleftarrow lineSearch(set)$
        if $thisInfoGain > infoGainBest$ then
            $attributeBest \longleftarrow attribute$
            $infoGainBest \longleftarrow thisInfoGain$
            $indexBest \longleftarrow$ index given by line search
            $thresholdBest \longleftarrow$ attribute value midway between $indexBest$ and $indexBest + 1$
    if not $infoGainBest$ or not $attributeBest$ then
        no attribute provides an information gain.
        return leaf with mode of $set(1)$ (class labels)
    re-sort set along attributeBest.
    $subsets \longleftarrow getSubsets(set, attributeBest, indexBest)$
    $subtree1 \longleftarrow trainDecisionTree(subsets(1), minLeaf)$
    $subtree2 \longleftarrow trainDecisionTree(subsets(2), minLeaf)$
    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.
begin
    for each sample $\in$ set do
        treeWalked $\longleftarrow$ tree
        classified $\longleftarrow$ false
        attributes $\longleftarrow$ features corresponding to this sample
        while not classified do
            if treeWalked is a node then
                thisAttribute $\longleftarrow$ attribute given by treeWalked
                thisAttributeValue $\longleftarrow$ value of the sample's feature at thisAttribute
                thisThreshold $\longleftarrow$ threshold given by treeWalked
                remove thisAttribute from the sample's feature vector.
                compare thisAttributeValue to thisThreshold:
                if thisAttributeValue $<$ thisThreshold then
                    choose left branch.
                    treeWalked $\longleftarrow$ left branch given by current treeWalked.
                else
                    choose right branch.
                    treeWalked $\longleftarrow$ right branch given by current treeWalked.
            else
                treeWalked is a leaf.
                append predicted label to set(2) at this sample.
                classified $\longleftarrow$ 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

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

Where $\mathrm{P}(x_i)$ 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, $\mathrm{P}(x_i) = 1$ and $\log_2 \mathrm{P}(x_i) = 0$, 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 [5]. This is better expressed by the following equation:

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

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.

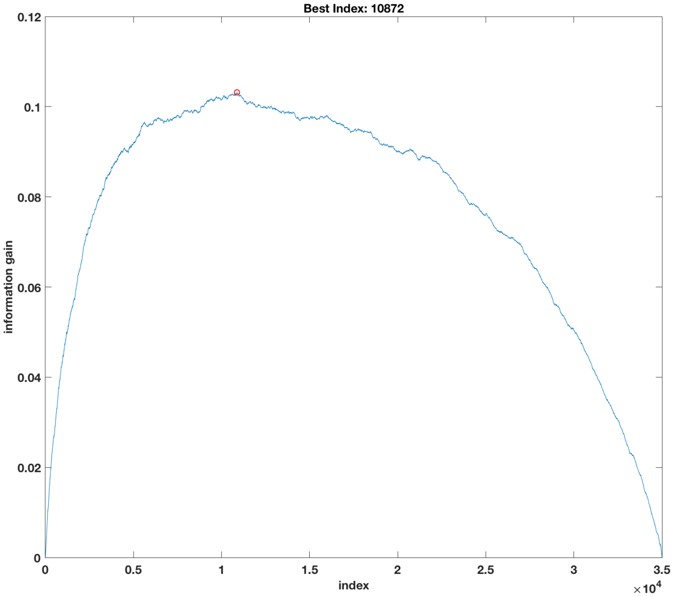


Figure 3.2: Information Gain across all possible thresholds.

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 [6].

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 [7].

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

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:
$\boldsymbol{n}$ is the number of classes.
$\boldsymbol{N}$ is the total number of training data samples.
$\boldsymbol{N}_i$ 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 $f_i = \boldsymbol{\mu}_i\Sigma^{-1}\boldsymbol{x}^T - \frac{1}{2}\boldsymbol{\mu}_i\Sigma^{-1}\boldsymbol{\mu}_i^T + ln(P(i))$. 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, extra-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 our best extra-tree result on MNIST, we split the dataset in half and trained 100 trees. The Yale B result comes from one of our 5-fold cross validations with 50 trees.

Algorithm	MNIST	Yale B
LDA	13.5%	6.8%
Decision Tree	16.6%	57.89%
Extra-Trees	4.89%	34.02%

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 stability of our classifiers irrespective of variations in the 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`. Future implementations could include more complex parameters. In k-fold cross validation, k classifiers are trained on different subsets of the entire set, ensuring that each sample of the dataset is used only once for validation. This helps to mitigate the effects of variation in the dataset and illustrate the general performance of the classifier at hand. The k classifiers' error rates are averaged to give a strong estimate of the algorithm's true performance. The figure below illustrates this process for $k = 4$.

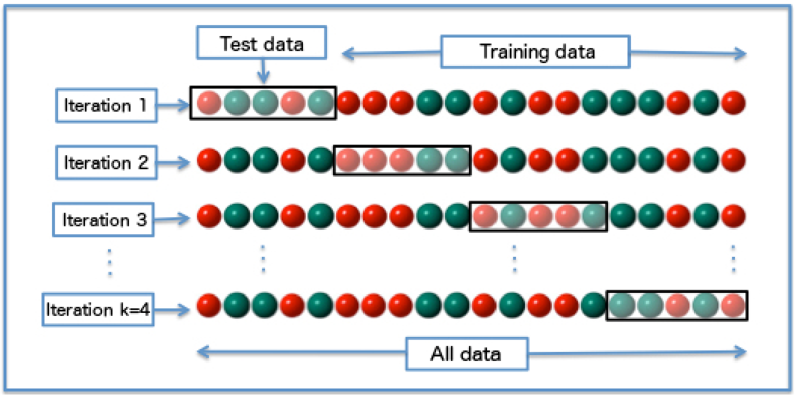


Figure 4.1: 4-fold cross validation on a dataset.

The best results of our cross validation are represented in the table below. The error rates are averaged over the 5 folds. Note that because the 5-fold cross validation yields a larger test set for Yale B than in previous results (§4), the error rate for decision trees on the Yale B dataset was higher. If we used a higher-fold cross validation, the results would have been better.

Algorithm	MNIST	Yale B
LDA	14.5%	2.5%
Decision Tree	17.9% (`minLeaf = 1`)	74.9% (`minLeaf = 1`)
Extra-Trees	5.4% (100 extra-trees)	36.3% (100 extra-trees)

Table 4.2: Average classification errors achieved with LDA, Decision Trees, and Extra-Trees after Cross Validation.

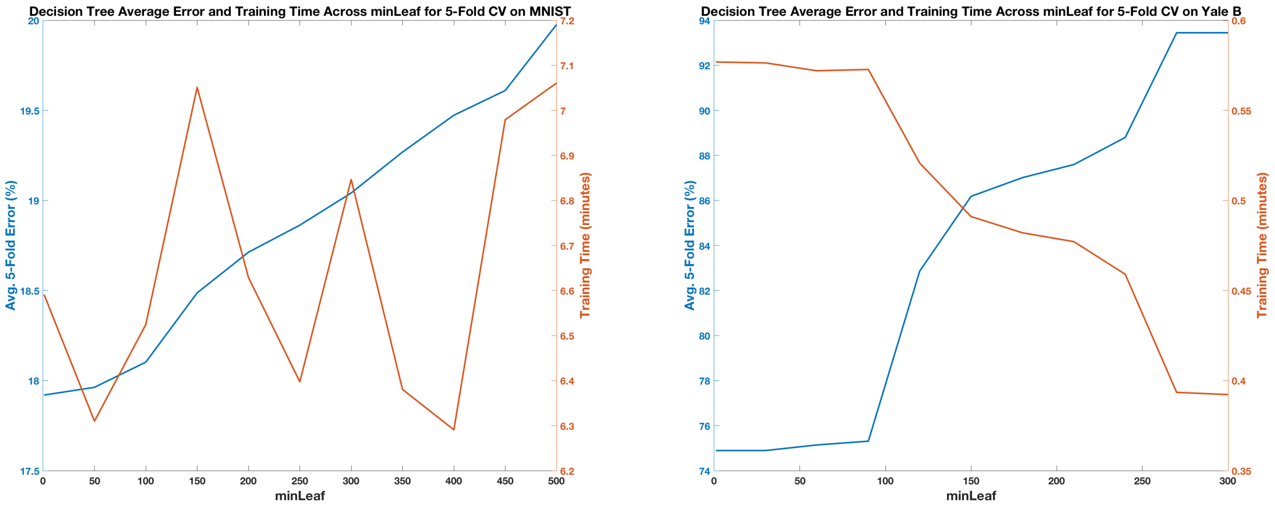
4.2 Hyperparameter Optimization

4.2.1 Decision Trees

The figure below represents the accuracy and time performance of our decision tree algorithm on MNIST and Yale B after averaging with cross validation. Using cross validation allows us to generalize the effects of our changing hyperparameter, `minLeaf`, which is a value that defines the minimum number of samples required to make a leaf node in the tree. As soon as the set is split to a level below `minLeaf`, a class label is applied as the *mode* of the samples in the set.

Note that the timing results for MNIST have high variability. This is because we ran our tests on a shared server that was at high occupancy. In this way, training our decision trees on MNIST was impacted by the other users. On our local computers, the decision tree was training in about 4 minutes, with a clear correlation between `minLeaf` and the time it took to complete. This process is evinced in the figure for the Yale B dataset, which was interrupted less on the server.

Our cross-validated results indicate that the highest accuracy is achieved by setting `minLeaf = 1`.



(a) (b)

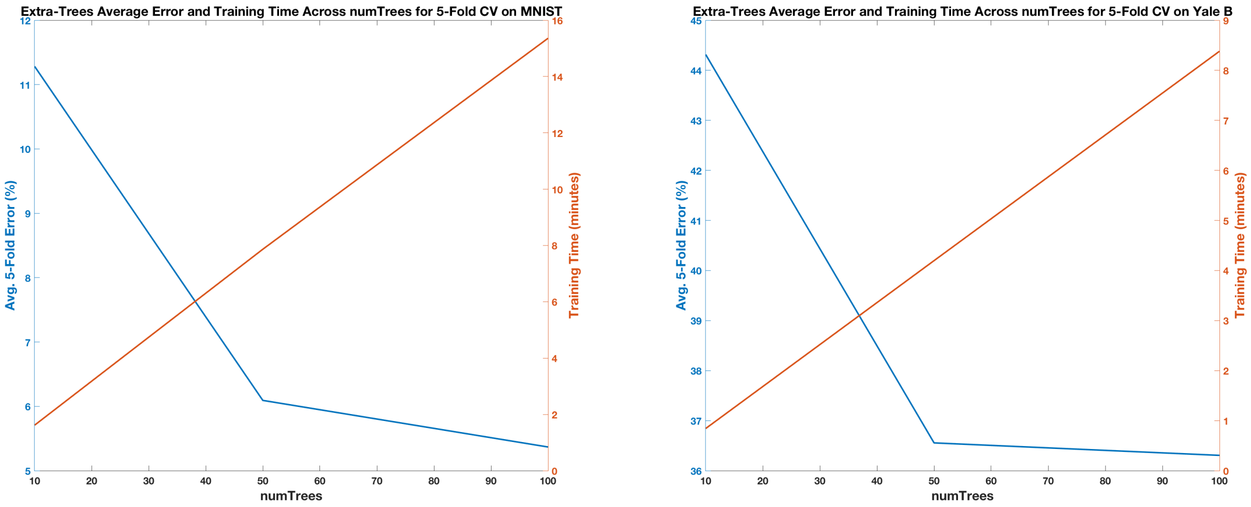
Figure 4.2: Cross Validated Decision Tree Hyperparameter Results on MNIST (a) and Yale B (b).

4.2.2 Extra Trees

The figure below represents the accuracy and time performance of our extra-tree classifier on MNIST and Yale B after averaging with cross validation. Using cross validation allows us to generalize the effects of our changing hyperparameter, `numTrees`, which is a value that defines the number of trees used for the ensemble. In general, more trees to vote on the classification label improves the results. Though a clear case of diminishing returns is observed after `numTrees = 50`.

Note that the time to train the extra-trees increases linearly with the number of trees, as the training of a decision tree, especially in the random case, is of constant time complexity. This does not include the testing of trees, which increases as $O(numTrees * \log m)$. This is derived by considering that $numTrees$ trees must be passed through to decide a class label, which takes $\log m$ time, where m is the depth of the tree.

Our cross-validated results indicate that the highest accuracy is achieved by setting `numTrees > 50`, with `numTrees = 100` being a good compromise between time to execute and accuracy.



(a)

(b)

Figure 4.3: Cross Validated Extra Tree Hyperparameter Results on MNIST (a) and Yale B (b).

4.2.3 LDA

Our error rate increased by 1% over our best result due to the increasing size of the training set from half to $\frac{4}{5}$ of the total data for 5-fold cross validation. This captures more of the noise and distortion in the images and thus is less performant. The figure below shows the variation among the 5 folds in error. As our error rates do not exhibit much variance, we feel the LDA model we used is stable.

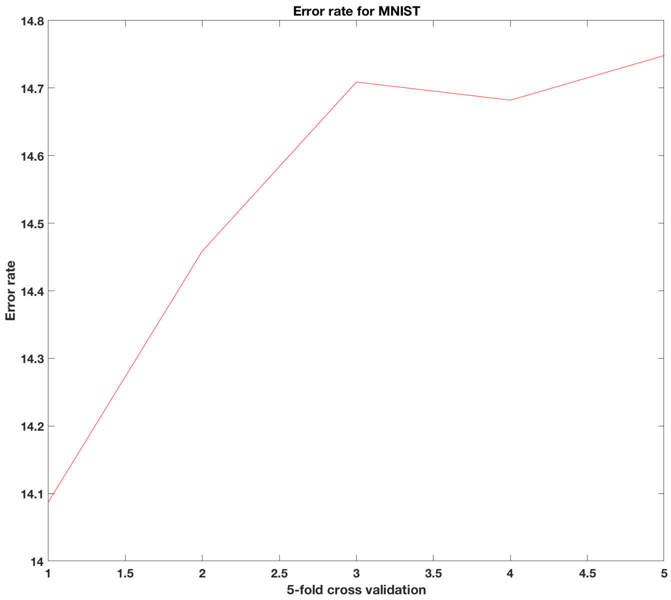


Figure 4.4: Cross Validated, LDA Error Rates for MNIST.

We also get a stable LDA model for the Yale B dataset. We increased our performance by increasing the training set and tested on different validation sets.

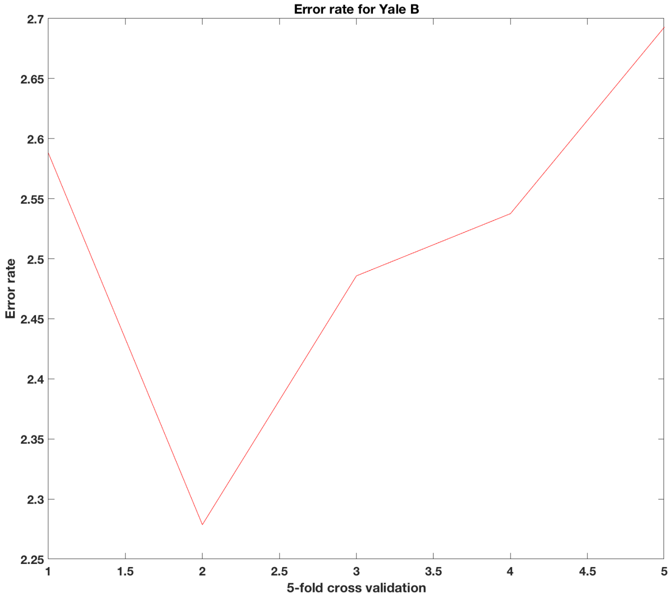


Figure 4.5: Cross Validated LDA Error Rates for Yale B.

Since LDA depends only on the mean vector and covariance matrice, there are no hyperparameters to consider. We tried to look at the prior probabilities to see if that increases our result, but sticking to equal probabilities yields a better result; this is intuitive as each class appearance is equiprobable.

5 Analysis of Classifiers

In this section we will discuss the advantages and drawbacks of each algorithm; additionally, we will discuss ways we feel would improve our results and the scalability of our classifiers.

5.1 Decision Trees

5.1.1 Advantages of Decision Trees

Some advantages of decision trees include [8].

1. Ability to handle both quantitative and qualitative data. Our data is purely numerical, but decision trees in general can support a variety of informative input types.
2. The hierarchy is easily represented in a graph form. The classifier is not a black box and can be fully understood. This is very appropriate for situations where the classifier's 'reasoning' needs to be transparent such that its intuitions (and biases) can be revealed.
3. Predicting a class label is executed in $O(\log m)$ time, where m is the depth of the tree.

4. Training data ordering is less impactful on the decision tree, as the data is being sorted at every available feature iteratively. In this way, the outcome of the decision tree classifier is most connected to the structure of the sets of data used, not their individual parts.

5.1.2 Disadvantages of Decision Trees

Some disadvantages of decision trees include [8].

1. There is a high computational complexity to training in our implementation. This is most evident in a large dataset where the first few levels of the tree split and subsequently sort a large body of data. Training requires iterating over all of the available features in the set at hand *and* searching over the possible splits to find the highest information gain.
2. Decision trees can become too complex if limits aren't placed on their growth. This can lead to overfitting. `minLeaf` serves to prevent this kind of action.
3. Single decision trees are sensitive to variations in data composition- feature generation is necessary as a single, classical decision tree cannot create a generalized classifier; it will also take very long to train if a raw image is used even of modest proportions.
4. Decision trees are susceptible to favoring the class mode of the dataset. Additionally, they require larger datasets and very discriminative features to be most accurate.

5.1.3 Improving Decision Trees

Much of the drawbacks of using decision trees can be mitigated by ensemble methods such as boosting, bagging, and random forests. These utilize stochastic factors that improve the classification power of a single, thorough decision tree by training many poorer decision trees that then vote on a classification. The majority vote is then chosen for the class label in testing. Ensembles of decision trees can also eliminate the need for preprocessing and feature generation. Our extra-tree implementation learns by considering only the raw pixel data. This eases the intellectual requirements to generate a capable classifier. Randomized decision trees train much faster than a classical decision tree, and our experiments show that they yield an improvement in accuracy, as well.

5.2 Extra Trees

5.2.1 Advantages of Extra Trees

Extra trees have nearly the same advantages as decision trees. Though, due to the ensemble of randomly assigned splitting criteria, extra trees have less 'reasonability' with respect to their decision making. The entirety of each tree is available, so they still are not black boxes, but their decision making is without any particular method. Additionally, the computational complexity of extra trees is less than that of classical decision trees. Though, taking the majority vote of all the decision trees introduces a logarithmic complexity on the order of the number of extra-trees employed.

Additionally, extra trees need no feature generation to be trained, they can run on the raw pixel data. Additionally, their training is less dependent on the size of the dataset and number of features it carries; in this way, extra trees scale well with the number of data samples. Note that there is $O(n)$ complexity in making a random split, where n is the size of the parent, unsplit set. This is derived from the comparison that must be made at the feature value. This is better than a classical decision tree, which needs to sort the set *and* calculate the information gain for each candidate split *for each* available feature.

5.2.2 Disadvantages of Extra Trees

Like classical decision trees, extra trees can also become too complex if limits aren't placed on their growth. Likewise, they may favor the class mode of the dataset.

Additionally, compared to classical decision trees, extra-trees require more work to label data, as the test sample must pass through every tree to obtain a vote. Then all of the votes are aggregated into an estimate. This means utilizing the classifier can be costly as the number of trees increases, precisely in $O(numTrees * \log m)$. This is derived by considering that $numTrees$ trees must be passed through to decide a class label, which takes $\log m$ time, where m is the depth of the tree.

5.2.3 Improving Extra Trees

A possible improvement on extra-trees is to utilize a sub-window technique to split each sample into subframes that are associated with the original class. This serves to augment the dataset and may be particularly useful in cases where there are a lack of samples available, as in the Yale B dataset.

In testing the data, a parallel computation can be utilized as each classifier votes independently of the others. In this way, the computational burden of testing the classifier can be reduced.

5.3 LDA

5.3.1 Advantages of LDA

Some advantages of LDA include [9].

1. The analytical solution makes the problem simpler and gives more insight into the statistics of the data.
2. The terms are all linear which eases the computational cost.
3. Well-suited to large data sets.

5.3.2 Disadvantages of LDA

Some disadvantages of LDA include [10].

1. LDA produces only $(C-1)$ feature projections, where C is the number of classes in the dataset. This limits the expressiveness of the features, especially in comparison to PCA.
2. Our implementation assumes Gaussian likelihoods, but if the true distributions are non-Gaussian, then LDA's projections may lose significant information.
3. LDA focuses more on the mean rather the variance, thus if information is more contained in the variance, we lose some of the expressiveness of the classes.

5.3.3 Improving LDA

We can improve our LDA implementation by considering:

1. Since LDA does not capture the rotation and translation of the raw images well, some deskewing methods may improve our classification results.
2. Covariance matrix smoothing can help to give a better feature generation through 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. We have also shown the results of cross validation performed on our datasets with our algorithms and their hyperparameters. This served to generalize the performance of our classifiers.

For a demonstration of our functions, please navigate to the `code` folder and execute the functions given by `demonstration_dt_MNIST.m` and others, where `dt` stands for decision trees and `et` stands for extra-trees. These demonstrations illustrate the entire process of partitioning our data, training our classifiers, and calculating the results.

7 References

[1] Sebastian Raschka. What is the difference between lda and pca for dimensionality reduction?, 2018. [Online; accessed 2018-03-14].

[2] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[3] Wikipedia contributors. Id3 algorithm — wikipedia, the free encyclopedia, 2017. [Online; accessed 14-March-2018].

[4] Wikipedia contributors. C4.5 algorithm — wikipedia, the free encyclopedia, 2018. [Online; accessed 14-March-2018].

[5] Wikipedia contributors. Information gain in decision trees — wikipedia, the free encyclopedia, 2017. [Online; accessed 15-March-2018].

[6] Tao Li, Shenghuo Zhu, and Mitsunori Ogihara. Using discriminant analysis for multi-class classification: an experimental investigation. *Knowledge and Information Systems*, 10(4):453–472, Nov 2006.

[7] Wikipedia contributors. Linear discriminant analysis — wikipedia, the free encyclopedia, 2017. [Online; accessed 15-March-2018].

[8] Scikit Learn. Scikit documentation - decision trees, 2018. [Online; accessed 2018-04-19].

[9] Petra Nass. Discriminant analysis, 2018. [Online; accessed 2018-04-20].

[10] Ricardo Gutierrez-Osuna. Linear discriminant analysis, 2018. [Online; accessed 2018-04-20].

8 Code Listings

Below are the demonstration scripts that execute the algorithms we implemented and print their results. Please see the `code` folder for supporting functions and scripts, including cross validation and data processing.

Listing 1: Demonstration Code for Decision Tree on MNIST.

```
%{

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);

        % Classification Error
        errors = nnz(test{2}(:,1) ~= test{2}(:,2));
        errorRate = (errors/N_te)*100;

        fprintf('\nnumFeatures: %d, minLeaf: %d, error rate: %2.2f\n', ...
                numFeatures, minLeaf, errorRate);

end
```

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

```
%{

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_et_YaleB()

        addpath('utility', 'YaleB', 'YaleB/data', ...
                'extraTree');

        fprintf('begin Yale B extra tree demonstration\n');

        % hyper-parameters
        N = 2414;
        N_tr = 2000; % training samples
        N_te = N - N_tr; % test samples

        % for decision tree
        minLeaf = 1; % to prevent overfitting
        numTrees = 50; % ensemble for majority voting

        % partition data
        [train, test] = loadYaleB(N_tr);

        % features are the raw pixels, so we reorder the cell
        train = {train{2}, train{1}};
        test = {test{2}, test{1}};

        %% train
        st = cputime;

        % create an ensemble of random trees
        trees = cell(numTrees, 1);
        for tree = 1:numTrees

                fprintf('tree: %d\n', tree);
                trees{tree} = trainExtraTree(train, minLeaf);

        end

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

        %% test
        st = cputime;

        test = testExtraTrees(test, trees);

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

        % Classification Error
        errors = nnz(test{1}(:,1) ~= test{1}(:,2));
        errorRate = (errors/N_te)*100;

        fprintf('\nnumTrees: %d, minLeaf: %d, error rate: %2.2f\n', ...
                numTrees, minLeaf, errorRate);

end
```

Listing 3: Demonstration Code for LDA on MNIST.

```
clear; close all;
addpath('./utility', './MNIST', './MNIST/data', './MNIST/loadMNIST', ...
        './lda');

seed = 152039828;
rng(seed); % for reproducibility

% define parameters
N_tr = 35e3; % training samples
N_te = 35e3; % test samples
k = 10; % number of classes
% partition data

%{
        1/2 of the dataset should be for training
        the other for testing

        MNIST contains 70k examples
%}

[train, test] = loadMNIST(N_tr);
%% Construct scatter matrices and calculate within-class and between class
% covariance
mu = mean(train{1,1}, 2);
num_variables = size(train{1,1},1);
% Let's standardize the data;
%variance = var(train{1,1}, 0,2);
%train{1,1} = (train{1,1}-repmat(mu,1, 60000))./variance;

%
Si = zeros(num_variables); Sb = zeros(num_variables);
S_cov = zeros(num_variables);
for i = 0:k-1
    ind = (train{1,2} == i);
    N_i = sum(ind);
    x = train{1,1}(:, ind);
    mu_i = mean(x, 2);
    S_cov = S_cov + cov(x');
    Si = Si + (1/N_tr)*(x - (repmat(mu_i,1, N_i)))*(x - (repmat(mu_i,1, N_i)))';
    Sb = Sb + (N_i/N_tr)*(mu_i - mu)*(mu_i - mu)'; % (1/k)
end

% We apply singular value decomposition in order to find eigenvalues and
% eigenvectors
[U D V] = svd(pinv(Si)*Sb); % lets try S_cov/k instead of Si; but it is the same result
a = [];
for i = 1:(k)
    a = [a D(i,i)];
end

% from here we can see that we only have 9 highest values as we expected

% We transform the training and testing data to a subspace
transf_matrix = U(:,1:(k-1));
transf_train = train{1,1}'* transf_matrix;
transf_test = test{1,1}'*transf_matrix;
% We find mean vector and covariance matrix for each class
mu_each_class = zeros(k-1, k);
cov_each_class = {};
sum_cov = zeros(k-1,k-1);
for i =0:k-1
    ind = find(train{1,2} == i);
    X = transf_train(ind, :);
    mu_each_class(:, i+1) = mean(X, 1)';
    cov_each_class{1, i+1} = cov(X);
    sum_cov = sum_cov + cov(X);
end
% since we assume equal covariance in all classes, we take the average of
% covariance matrices
average_cov = sum_cov/k;
cov_equal_each_class = {average_cov average_cov average_cov average_cov average_cov average_cov average_cov average_cov average_cov average_cov};

[acc_test acc_train] = classify_comparison_same_cov(k,5,mu_each_class, cov_each_class, average_cov, ...
transf_test', test{1,2}, transf_train', train{1,2}); % 0.88 and 0.89 resp using just knn
%[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
```

Listing 4: Demonstration Code for LDA on Yale B.

```
clear; close all;
addpath('./utility');
addpath('./YaleB', './YaleB/data', './YaleB/50Train', './decisionTree');
%%
[train, test] = loadYaleB();
train{1,2} = train{1,2} - 1;
```

```
test{1,2} = test{1,2} - 1;
k = 38;
N_tr = size(test{1,2},1);
%% Construct scatter matrices and calculate within-class and between class
% covariance
mu = mean(train{1,1}, 2);
num_variables = size(train{1,1},1);
% Let's standardize the data;
%variance = var(train{1,1}, 0,2);
%train{1,1} = (train{1,1}-repmat(mu,1, 1900))./variance;
%
Si = zeros(num_variables); Sb = zeros(num_variables);
for i = 0:k-1
    ind = (train{1,2} == i);
    N_i = sum(ind);
    x = train{1,1}(:, ind);
    mu_i = mean(x, 2);
    Si = Si + (1/N_tr)*(x - (repmat(mu_i,1, N_i)))*(x - (repmat(mu_i,1, N_i)))';
    Sb = Sb + (N_i/N_tr)*(mu_i - mu)*(mu_i - mu)'; % (1/k)
end

% We apply singular value decomposition in order to find eigenvalues and
% eigenvectors
[U D V] = svd(pinv(Si)*Sb);
%% We transform the training and testing data to a subspace
transf_train = train{1,1}'* U(:,1:(k-1));
transf_test = test{1,1}'*U(:, 1:(k-1));
% calculting the multivariate parameters
mu_each_class = zeros(k-1, k);
cov_each_class = {};
sum_cov = zeros(k-1,k-1);
for i =0:k-1
    ind = find(train{1,2} == i);
    X = transf_train(ind, :);
    mu_each_class(:, i+1) = mean(X, 1)';
    cov_each_class{1, i+1} = cov(X);
    sum_cov = sum_cov + cov(X);
end
average_cov = sum_cov/k;
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};
% to see how our model works
[acc_test_comp acc_train_comp] = classify_comparison_same_cov(k,5,mu_each_class, cov_each_class,average_cov, transf_test', test{1,2}, transf_train', train{1,2}); % 0.88 and 0.89 resp using just knn
```