Project Part 1
Pattern Recognition
ECE 759

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

March 14, 2018

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

List of Figures

Listings

1 Introduction

2 Feature Selection

2.1 Decision Tree Feature Generation

2.1.1 MNIST

In working with the decision trees, we utilized the SVD of each image in the training set.

2.2 LDA vs PCA

LDA and PCA linearly transforms the data to reduce the dimensions. The difference is that LDA is supervised whereas PCA is unsupervised. PCA finds directions which maximize the variance along that direction and they are orthogonal to each other. We can view a PCA technique as in Figure(2.1).

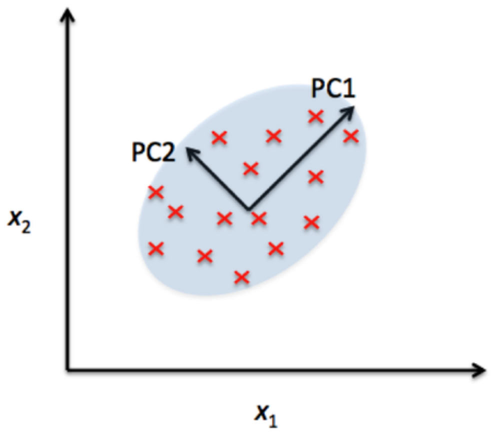

Figure 2.1: PCA implemented on 2-dimensional data

LDA maximizes the class separability and can be viewed as in Figure(2.2).

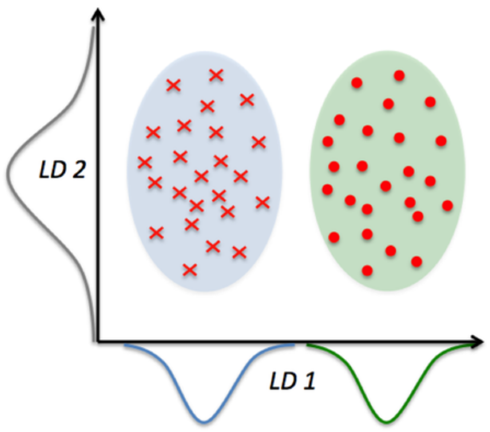

Figure 2.2: LDA visualization

The underlying concept of LDA is taking the eigenvectors of $\frac{\Sigma_b}{\Sigma}$ where Σ is within-class scatter matrix and Σ_b is between-class scatter matrix. By having this division of matrices, we separate the classes away from each other. On the other hand, since PCA is unsupervised it only tries to capture the data in minimum number of variables.

2.3 LDA for MNIST and Extended YaleB

LDA reduces the data with K classes with large dimensions to K-1 dimensions which captures the most energy of the data. Implementing LDA as a dimension reduction, MNIST and Extended YaleB datasets are reduced to 9 and 37 variables for each data point. Then we worked with this transformed data to do further classification.

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.

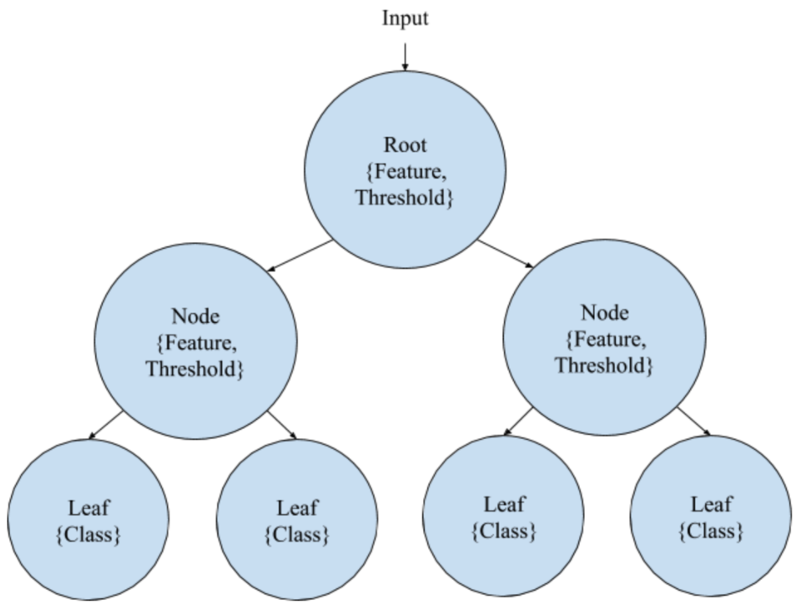

Figure 3.1: An example decision tree.

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

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

Where `set` is the training set, which is a MATLAB structure that contains the raw data (unused), 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 [1] [2].

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. 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.

   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 subsets given by `indBest` to find the next attribute that yields the highest information gain. Note that we exclude the attribute/feature we chose in this execution of `trainDecisionTree(.)`.

The algorithm is 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. *i.e.* a leaf is generated if the set has fewer elements than *minLeaf*.
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:
    $attributeBest \longleftarrow 0$
    $thresholdBest \longleftarrow 0$
    $infoGainBest \longleftarrow 0$
    $indexBest \longleftarrow 0$
    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. 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 ∈ set do
        treeWalked ⟵ tree
        classified ⟵ false
        attributes ⟵ features corresponding to this sample
        while not classified do
            if treeWalked is a node then
                thisAttribute ⟵ attribute given by treeWalked
                thisAttributeValue ⟵ value of the sample's feature at thisAttribute
                thisThreshold ⟵ threshold given by treeWalked
                remove thisAttribute from the sample's feature vector.
                compare thisAttributeValue to thisThreshold:
                if thisAttributeValue < thisThreshold then
                    choose left branch.
                    treeWalked ⟵ left branch given by current treeWalked.
                else
                    choose right branch.
                    treeWalked ⟵ right branch given by current treeWalked.
            else
                treeWalked is a leaf.
                append predicted label to set(2) at this sample.
                classified ⟵ true
```

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 topic 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. 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. 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 data-set, 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.

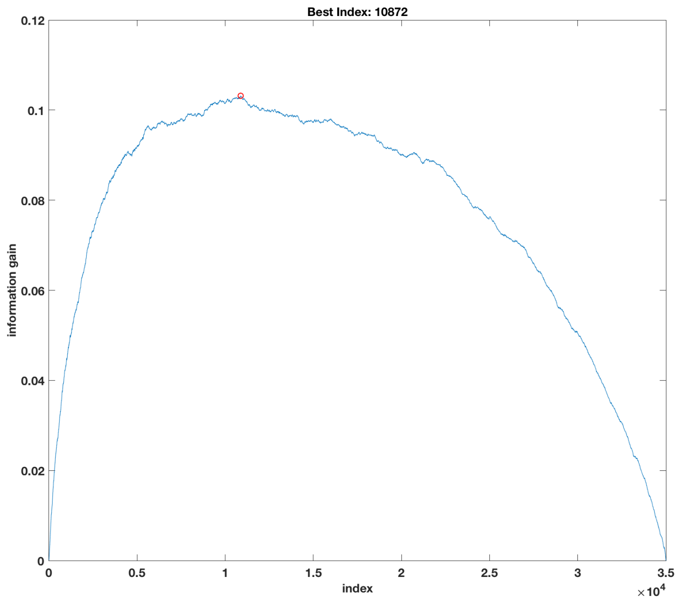

Figure 3.2: Information Gain across all possible thresholds.

3.2 Linear Discriminant Analysis

Our classification criterion is to misclassify as small as possible. The rule we employ in classifying the data points is through the use of Bayes Rule. We put the data point to the group with the highest conditional probability(i.e. $P(w_i|x)$). In practice, it is not feasible to get conditional probability for a given point unless we have a huge data. So we should assume the distribution and calculate the probabilities.

LDA relies on the assumption of normal distribution of data for each class. Linear discriminant analysis frequently achieves good performances in the tasks of face and object recognition, even though the assumptions of common covariance matrix among groups and normality are often violated (Duda, et al., 2001) (Tao Li, et al., 2006)'.

Since MNIST and Yale datasets are high-dimensional, we can not check the normality of variables. Instead, we can reduce the dimensions via a projection. We need to employ multiclass LDA as we have several classes. The class separation in this case will be given by the ratio of $\frac{\boldsymbol{w}^T\Sigma_b\boldsymbol{w}}{\boldsymbol{w}\Sigma\boldsymbol{w}}$. In the case of two classes, this will reduce just to the ratio of between-class variance to within-class variance.

3.2.1 Algorithm steps

Variable notations:
$\boldsymbol{n}$ is number of classes
$\boldsymbol{N}$ is the total number of training data
$\boldsymbol{N}_i$ is the number of points in each class
$\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}$ are mean vectors for each class and overall mean for the data

The steps that we follow in our LDA algorithm are:

1. We calculate within-class scatter matrix $\Sigma_i = \frac{1}{N_i - 1} \sum_{\boldsymbol{x} \in D_i}^{n} (\boldsymbol{x} - \boldsymbol{\mu}_i)\,(\boldsymbol{x} - \boldsymbol{\mu}_i)^T$ for each class, then sum them up to get $\Sigma_W = \sum_{i=1}^{c} (N_i - 1)\Sigma_i$.
2. Then we find average within-class scatter matrix by calculating $\Sigma = \dfrac{\Sigma_W}{N}$
3. We also calculate the between-class scatter matrix by $\Sigma_B = \sum_{i=1}^{c} \dfrac{N_i}{N} (\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^T$.
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 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. A discriminant function for each class is $f_i(x_k) = \mu_i {w_a}^{-1} {x_k}^T - \dfrac{1}{2} \mu_i {w_a}^{-1} {\mu_i}^T + ln(P_i)$ is evaluated where P_i is the probability of each class.
10. Then we select a class with highest discriminant function.

3.2.2 Helpful Functions

To accomplish the above pseudo code, we used own created functions.

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.

Algorithm	MNIST	Yale B
LDA		
Decision Tree		

4.1 Preliminary Hyperparameter Optimization

4.1.1 Decision Trees

The table below illustrates some results that we obtained during various configurations of the decision tree training algorithm. In an effort to optimize training time vs. test accuracy, we noticed some trends in the algorithm's performance. Notably, utilizing `minLeaf` doesn't seem to contribute to over-fitting, as was noted in other implementations [3].

Training Configuration	1	2	2	2	3	3	1	1	1	1	4
numFeatures	30	30	60	200	200	200	30	100	100	20	10
minLeaf	1	1	2	2	3	4	4	4	1	1	1
Error Rate	22	26	26	26	23	24	22.6	23.6	23.5	22.9	18.42
Mins To Train	48	2	2	3	12	12	47	60	102	11	3

5 Conclusion

In conclusion, we have shown that utilizing Decision Trees and Linear Discriminant Analysis 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.

6 References

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

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

[3] Mathworks. Matlab documentation - fitctree. https://www.mathworks.com/help/stats/fitctree.html, 2018. [Online; accessed 2010-09-30].

7 Code Listings

Below are the primary scripts that solve the project. Please see the `code` folder for supporting functions.