Project Part 1
Pattern Recognition
ECE 759

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

March 13, 2018

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

1 Introduction **2**

2 Feature Selection **2**

2.1 Decision Tree Feature Generation 2

2.1.1 MNIST 2

3 Algorithm Implementations **2**

3.1 Decision Tree Algorithm 2

3.1.1 Information Gain and Entropy . . 4

4 Algorithm Implementations **4**

4.1 Linear Discriminant Analysis 4

5 Code Listings **5**

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 in the training set.

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 attribute of the test data is compared against. If the test attribute is less than the threshold, we proceed down the left branch. Otherwise, the right.

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, an attribute and threshold are specified. If a test sample's feature at that particular attribute 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 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.

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. 3.1), 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 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 $!thisSetEntropy$ 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 $attribute \in range(\#offeaturesleftinset)$ do
        sort $set$ along $attribute$.
        perform line search heuristic to approximate max information gain by splitting $set$ at various indices.

        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 $!infoGainBest$ or $!attributeBest$ 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 $subtree2$
```

3.1.1 Information Gain and Entropy

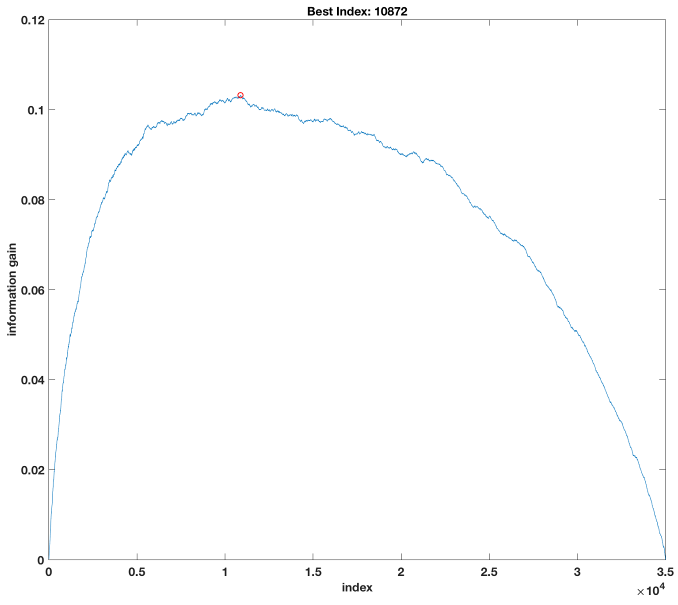

Figure 3.1: Information Gain across all possible thresholds.

4 Algorithm Implementations

4.1 Linear Discriminant Analysis

Our classification criterion is to misclassify as small as possible. The rule we employ in classifying the data points is through use of Bayes Rule. We put the data point to the group with the highest conditional probability. 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 10 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.

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. We also calculate the between-class scatter matrix by $S_B = \sum_{i=1}^{c} N_i(\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^T$.
3. We need to find eigenvectors and eigenvalues of $\Sigma^{-1}\Sigma_b$.
4. We then sort the eigenvectors depending on the magnitude of eigenvalues.
5. The number of linear discriminants will be $c - 1$ which will be 9. We need to verify that.
6. Project our data onto the subspace(constructed by the eigenvectors of the highest eigenvalues).
7. Since we have a reduced dimension for our data, we can easily apply nearest neighbors method in order to get the classes for our test data
8.

5 Code Listings

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