ECE 759 Project 1 Part 1

Matt Conrad and Evan Williams

April 22, 2018

Abstract

In this report we propose a feature extraction and feature selection scheme for the classification of images from two datasets: MNIST and Caltech-10. Following this, we describe the implementation of two algorithms, Linear Discriminant Analysis and Decision Tree Classification, of which the source code can be found in the accompanying MATLAB files. Example results on the Iris dataset using a random 50% training/testing split yield a classification accuracy of 94.67% on the test set for the decision tree classifier with hyperparamaters MaxSplits = 10, StopCriteria = 0.01, MaxDepth = 5 while the LDA classifier had a classification accuracy of 97.33% on the test set. The script used for this evaluation can be found in the submission as test_script.m.

1 Introduction

The fields of machine learning, image processing, and computer vision are continually growing and becoming further intertwined to create state-of-the-art technology, such as real-time facial recognition. As these technologies become increasingly complex, it is essential to maintain an understanding of basic algorithms and techniques that the complex techniques rely on.

The two fundamental machine learning algorithms covered here include Linear Discriminant Analysis (LDA) and Decision Trees. Both are a form of classifier that require supervised training in order to classify new feature vectors into one class or another. While they do aim to perform the same task, LDA classifies linearly whereas the Decision Tree does so nonlinearly [TK09]. Their respective training algorithms and the math behind them will be explained in further detail in their own subsections.

Although these classifiers are useful tools in themselves, they require feature datasets for training and testing. The raw data for this project was presented in the form of raw images in the publicly available MNIST and Caltech10 image sets. In this form, image processing and computer vision algorithms were needed for transforming the raw images into usable feature vectors. The techniques used for feature extraction will be described in a future subsection.

2 Image Sets

The data being considered was supplied as the image sets MNIST and Caltech10. MNIST is a collection of 60,000 images of handwritten digits (0-9). Each MNIST image is a grayscale 28x28 sized image with 8-bit intensity levels. In contrast, the Caltech10 image set has 646 images of 10 different types of objects, including ants, butterflies, cameras, and other distinctly shaped objects. This data set is much more dynamic in that it contains both grayscale and RGB-colored images, where the images are not guaranteed to be the same size. These properties require the feature extraction process to implement considerations to protect from the variations in the Caltech10 data set.

To train and test the classifiers presented here, we split each image set so that one half of the images of a class were in the training set and the other half were in the testing set. This led to MNIST having training and testing set of 30000 images each, whereas Caltech10 had 325 images in the training set and 321 images in the testing set.

3 Feature Extraction

The problem of feature extraction from images can be considered separately from the classifier problem, because classifiers only see numbers and do not care where the data comes from. In addition, MNIST and Caltech10 require two separate feature extraction processes due to the differences in the types of images being considered.

MNIST being more uniform and simpler than Caltech10, could have a feature vector as simple as being just the pixel values. The size of this particular feature vector came out to be 784 (28x28) features. While extremely simple, this feature vector choice may not contain sufficient information. For example, say there are two MNIST images of the number 4. Figure 1 gives such an example in the form of a binary "image" where colors are used for illustration purposes. In this example, one person writes 4 large, rectangular, and thin-lined, whereas another person writes 4 to be compact, triangular, and heavy-lined. It is also not out of the question that one person writes their number slightly less centered. This leads to a situation where the markings only overlap by 4 pixels out of the 400 in this "image". Thus, if these images were points in the feature vector space they would be quite far away despite being the same number.

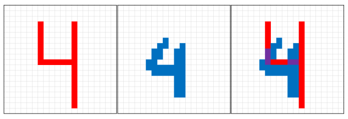

Figure 1: Different ways of writing the number 4 as a binary "image"

In an effort to more accurately describe the information in the MNIST image set, a feature descriptor was employed. In image description, the most common categories of characteristics described are color, texture, and shape. MNIST, being grayscale and simple, only allowed for shape feature descriptors to be used. Of this class of descriptors, Histogram of Oriented Gradients (HOG) was determined to be a good choice since it provides a more sophisticated and global approach to image description over raw pixel values.

The HOG algorithm starts by computing the gradient values of the image; this step can be accomplished by convolving a horizontal 1-D derivative kernel with the image to get the x-component of the gradient, then a vertical kernel to get the y-component. The gradient image then can be broken into a number of non-overlapping squares called cells. Each cell has the same number of pixels. The next step involves having each pixel cast a vote for an orientation histogram for its particular cell. Each vote is weighted based on the magnitude of the computed gradient and is casted into the bin based on the computed gradient's orientation. This histogram usually contains 9 bins ranging from 0 to 180 degree-values. To account for changes in illumination and contrast, the algorithm then groups adjacent cells into larger groups called blocks. All cell orientation histograms in a block are normalized to the block. These blocks often overlap so that a cell can be included and normalized by multiple blocks. Once normalization has finished, the final step is to concatenate all blocks into a 1-D feature vector [DT05].

HOG is a dense technique, meaning it calculates evenly across the image and not sparsely as in other shape-based feature descriptors like SIFT. Such a characteristic allows for a consistent and controllable number of features in the feature vector, which is ideal for machine learning. It would be difficult to compare data points that have different sizes of feature vectors. This descriptor is also invariant to geometric and photometric transformations making it more robust to data sets like MNIST and Caltech10 that have significant variance in the shape of the object. Figure 2 illustrates how a HOG algorithm captures shape information in the orientation histograms of cells.

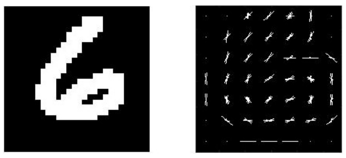

Figure 2: An example of the HOG visualization showing orientation histograms in their respective cells.

In the more challenging case of the Caltech10 image set, we must be a little more careful when extracting features. Before doing that, we wrote a custom script that organized the images into a data structure that contained the class label, image size, and image data. The image data from this structure was then used by the feature extraction methods described next.

The RGB images in Caltech10 must be converted to grayscale since (1) several images are already grayscale and (2) some images are not colored conventionally. An example of this is seen in Figure 3. The utilization of color feature descriptors would not be useful here since colors of Caltech10 objects are not always consistent.

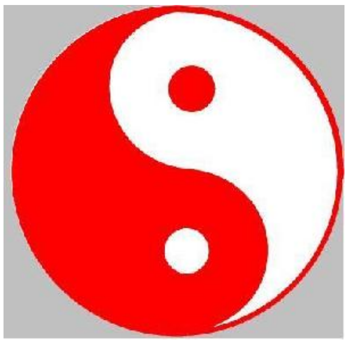

Figure 3: Image from Caltech10 image set of red-white yin-yang symbol, which differs from many others which are black and white

Similar to color information, it appears texture feature descriptors such as Local Binary Patterns (LBP) are also not useful in this situation because images of the same object come from many different sources. Figure 4 shows that two images can have completely different textures.

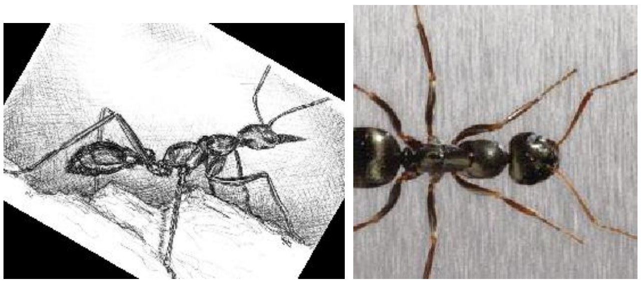

Figure 4: Images from Caltech10 image set of two ants. One is sketched while one is real leading to two completely different object textures

This leaves shape descriptors, such as HOG, left. We saw HOG provided useful shape information in MNIST and can be similarly applied to Caltech10. A weakness of HOG is that it does not handle changes in object orientation as well as others like SIFT [DT05]. Luckily, Caltech10 classes have similarly oriented objects such as in Figure 4 where both ants are aligned horizontally. Thus, HOG suits both image sets well.

Now that a descriptor algorithm has been chosen to create the feature vector, it is necessary to optimize HOG parameters for a well-constructed feature vector. To do this, we varied the cell size parameter to change the composition and size of the feature vector. HOG, being a shape-oriented descriptor, can be visually inspected for quality of shape encoding [DT05]. As seen in Figure 5, a good choice (4x4 in this case) of cell size can be used to encode the shape information of the number 2 without using too many features. Thus, we incrementally decreased the cell size to the point where shape information first became visible.

For the MNIST data set, the cell size (in pixels) was chosen to be 4x4 as seen in Figure 2. Any larger cell size either caused partial coverage of the 28x28 grid or created visually poor descriptors.

In contrast, the Caltech10 feature vector was calculated so that the image was divided into a 5x5 cell array as seen in 6. By specifying the cell array and not cell size here, we effectively solved the issue of varying image size seen in the Caltech10 image set. Again, these cell size parameters for the two data sets were determined visually. Improvements to these parameters must be explored while conducting performance analysis, which will be done in the next installment of this study.

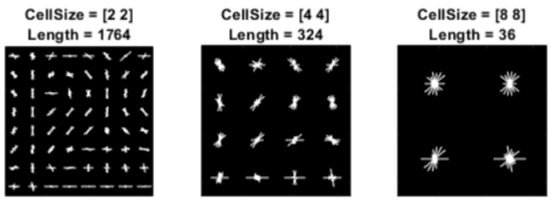

Figure 5: Encoding shape information of the number 2 in HOG feature vectors

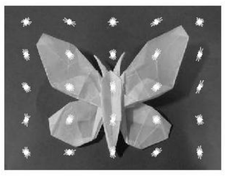

Figure 6: HOG descriptor overlain on a Caltech10 butterfly image

Algorithm 1 provides the pseudocode used to extract the features from the raw images. This pseudocode utilizes the MATLAB extractHOGFeatures function in the Computer Vision toolbox to conduct the HOG algorithm outlined earlier.

Algorithm 1: Feature Extraction

```
1 function Extraction ($ImageSubset, NumberOfCellColumns, NumberOfCellRows$);
  Input  : An ($ImageSubset$) (either Train or Test) as well as the parameters
           ($NumberOfCellRows$) or ($NumberOfCellColumns$) for defining the cell array
  Output: The 1-D feature vector ($FeatureVector$)
2 Allocate space for an MxN feature array, where M is the number of training points in
    ($ImageSubset$) and N is the length of the feature vector
3 If the image set is of Caltech10, then convert RGB to grayscale
4 for each image do
5     Store the image size as $m$ rows and $n$ columns
6     Calculate the dimensions of HOG cells to be
        $Floor(\frac{m}{NumberOfCellRows})$ rows and $Floor(\frac{n}{NumberOfCellColumns})$ columns
        where $Floor$ is the function that rounds down to the nearest integer
7     Perform the HOG feature extraction by passing the current grayscale image along with
        the cell size parameters to MATLAB's $extractHOGFeatures$ function. Store the
        resulting 1-D feature vector in the allocated feature array.
8 end
```

4 Algorithmic Implementation

Two classifiers were implemented in the first part of the project. Both classifiers were implemented using MATLAB.

4.1 Linear Discriminant Analysis

Linear Discriminant Analysis can be used both as a classifier as well as a dimensionality reduction technique [HTF09]. The basic assumption of this classification technique is that the data of each class is distributed as a multivariate Gaussian as seen in equation 1.

$$f_k(x) = \frac{1}{\sqrt{(2\pi)^{(p)}|\mathbf{\Sigma}_k|}} \exp\left(-\frac{1}{2}(x-\mu_k)^T {\mathbf{\Sigma}_k}^{-1}(x-\mu_k)\right) \quad (1)$$

From the use of prior probabilities (Let π_k be the prior probability of class k) and Bayes' Rule the classification problem can thus be formulated as seen in equation 2.

$$P(Class = k|X = x) = \frac{f_k(x)\pi_k}{\sum_{i=1}^{K} f_i(x)\pi_i} \tag{2}$$

Linear Discriminant Analysis arises in the special case where we assume the classes have a common covariance matrix $\hat{\Sigma} = \Sigma_k \forall k$ [HTF09]. In this case the log likelihood of the classes k and i yields linear discriminant coefficients for each class seen in equation 3 which is the bias term as well as in equation 4 which is a vector containing the coefficients corresponding to the number of features in the data. A discriminant function is created for each class in the data.

$$\delta_{k0} = \frac{-1}{2}\mu_k^T \hat{\Sigma}^{-1}\mu_k \tag{3}$$

$$\delta_{kN}(x) = \hat{\Sigma}^{-1}\mu_k \tag{4}$$

Finally, the discriminant functions can be evaluated with a scoring function which takes into account the priors, bias term, and learned discriminant coefficients as seen in equation 5.

$$\hat{s}_k^L(x) = \delta_{k0} + \sum_{kj}^{N} \delta_{kj} x_j + log(\hat{\pi}_k) \tag{5}$$

The assigned class of test data is simply the highest scored class as calculated by the scoring function above [HTF09].

To train this model you need to estimate three parameters within the training data: class prior ($\hat{\pi}_k$), class mean ($\hat{\mu}_k$), and the pooled covariance matrix $\hat{\Sigma}$. Equations 6, 7, and 8 respectively show the computations required to estimate each parameter.

$$\hat{\pi}_k = \frac{N_k}{N} \tag{6}$$

Where N_k is the number of class-K observations.

$$\hat{\mu}_k = \sum_{g_i = k} \frac{x_i}{N_k} \tag{7}$$

$$\hat{\Sigma} = \frac{\sum_{k=1}^{K} \sum_{g_i=k} (x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T}{N - K} \tag{8}$$

Now that we have delved into how the classifier works we can delve into the specific implementation written in MATLAB. This algorithm is implemented in a function named LDA and takes in three inputs: TrainFeatures, TrainLabels, and TestFeatures. TrainFeatures is a matrix of size M1xN where M1 is the number of training data images and N is the number of training data features. TrainLabels is a matrix of size M1x1 where M1 is the number of training images. TestFeatures is a matrix of size M2xN where M2 is the number of testing data images and N is the number of features. The function returns a single output TestPredictions which is a matrix of size M2x1 containing the predicted class for each test image. TestPredictions = LDA(TrainFeatures, TrainLabels, TestFeatures) is an example usage of the function call within MATLAB. Other usage cases of the LDA function can be found within the ECE759_Project.m script.

Next, let us delve into the LDA function by providing pseudocode of its functionality and highlighting key steps. The pseudocode is seen below in Algorithm 2. For clarity we should note that the pooled covariance matrix is initialized as a zero matrix of size N x N. This parameter is updated as the iterative sum of each class's covariance matrix which is then normalized by Bessel's correction (N-K) at the end of the for loop. All other parameters should be sufficiently clearly explained in the pseudocode and relevant equations.

Algorithm 2: Linear Discriminant Analysis for multi-class classification

```
1  function LDA ($TrainFeatures, TrainLabels, TestFeatures$);
   Input   : Feature Matrices ($TrainFeatures$) and ($TestFeatures$) as well as Class Labels for
             Training Data ($TrainLabels$)
   Output: Predicted Class Labels of Testing Data ($TestPredictions$)
2  First, determine the number of unique classes ($K$) using the Training Labels.
3  for each class k to class K do
4      Calculate the class mean $\hat{\mu}_k$, pooled covariance $\hat{\Sigma}$, and class prior $\hat{\pi}_k$ using equations 6-8
5  end
6  Now that the parameters have been estimated for each class we can go through and calculate
     the discriminant functions for each class. To clarify we note that the pooled covariance
     matrix is initially a zero matrix of size which is iteratively updated within the first for loop.
7  for each class k up to class K do
8      Calculate the discriminant function for each class k using equations 3 and 4.
9  end
10 Next, evaluate the testing data using the learned model.
11 for testing point i in M2 testing points do
12     for each class k up to class K do
13         Calculate the likelihood of the datapoint i belonging to class k as seen in equation 5
14     end
15     Predict each testing point i as the class with the argmax of the likelihood scoring
16 end
17 Finally, return the vector of M2 predictions of the testing data as your output.
```

4.2 Decision Tree Classifier

Decision Tree Classifiers are a classification technique that partition the data into ever smaller nodes based on a best splitting rule that maximizes the separation amongst classes. There are several methods that can be employed to grow Decision Trees such as C4.5 and ID3. The method that we employed is similar to ID3 in that it is a greedy technique which attempts to optimize the sub problem of maximization of separation amongst data using the best found split however we use a different cost function. We utilize the Gini Index which yields a value from 0-1 which describes the impurity of each node. If a node contains only one class it is pure and has a Gini Index of 0 whereas an impure node with many classes of unequal frequency will yield a value closer to 1. Once a tree has been built it employs a simple thresholding rule to determine how you classify new data as seen in figure 7.

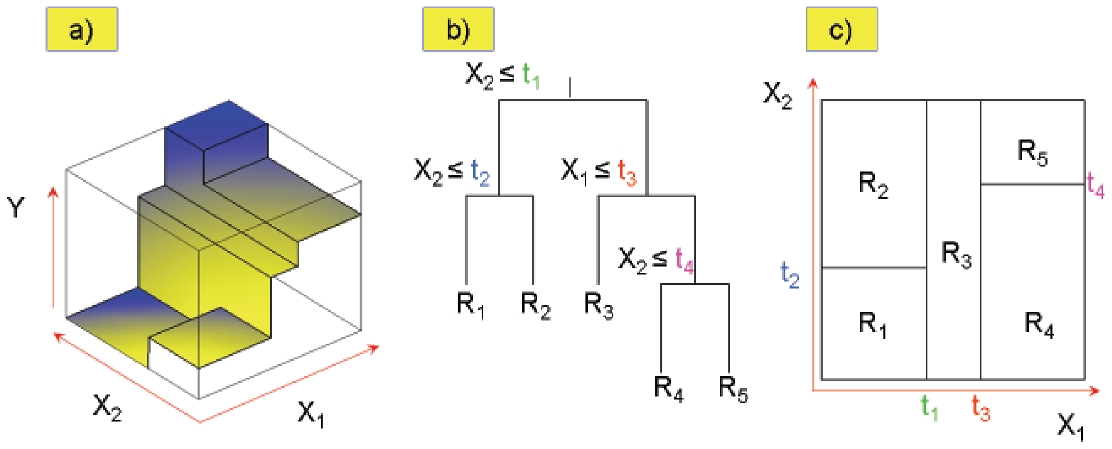

Figure 7: An example of a generated Decision Tree in various views [HTF09]

Now let's delve into the mathematics behind how to build a tree. First, we must describe the notion of node impurity. Equation 9 shows the definition of impurity where N_m is the number of training data points in node m, R_m is the region represented in node m, and I is the indicator function returning 1 if true else 0.

$$\hat{p}_{mk} = \frac{\sum_{x_i \in R_m} I(y_i = k)}{N_m} \tag{9}$$

From this basic definition the definition of the Gini Index arises in equation 10 where K is the number of classes in the data. This equation thus yields an expression of the homogeneity of the data in a particular node m.

$$GiniIndex = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{10}$$

Now that we have provided a cost function for the basis of evaluating node impurity we can begin to discuss how to consider growing the tree. To grow the tree we must attempt to maximize the homogeneity of each terminal by using the Gini Index as a measure to do so. The best split for a terminal node thus takes the form of iteratively looking at a certain split point s within the data in that node and a certain feature j to split the data. Each of these points is evaluated and the cost is assigned to be the average of the Gini Index of the proposed left and right child nodes. Equation 11 describes this technique mathematically where R_1 is the region containing data points split to the right $(x_i(j) \geq s)$ and R_2 is the region containing data points split to the left $(x_i(j) < s)$.

$$j^*, s^* = \min_{j,s} \frac{(GiniIndex(x_i \in R_1(j,s)) + GiniIndex(x_i \in R_2(j,s)))}{2} \forall j, s \tag{11}$$

The minimum of these scores is then assigned to be the proposed split of the data such that the threshold and feature of the node are assigned at split point j^* and s^* and two new child nodes are generated based on these values. The algorithm we employ also takes into account 3 hyperparameters which adjust the depth the tree can grow, the maximum number of splits that can occur when growing the tree, and the stopping criteria for when the tree generation process has converged to a stable solution. With that said let us now move into the pseudocode for the implementation in Algorithm 3. To elaborate on some aspects of the psuedocode which may not be clear the TreeDataStructure contains a number of Node structures. Each node struct contains information regarding the parent of said node, the child node to the right, the child node to the left, the depth of the node, the terminal status of the node (0/1), the members associated with the node, the most prevalent class in the node, the threshold of the node, and the feature that is being evaluated.

Algorithm 3: Decision Tree for multi-class classification

```
1  function GreedyDecisionTree
    ($TrainFeatures$, $TrainLabels$, $TestFeatures$, $MaxSplits$, $StoppingCriteria$, $MaxDepth$);
   Input  : Feature Matrices ($TrainFeatures$) and ($TestFeatures$) as well as Class Labels for
            Training Data ($TrainLabels$). ($MaxSplits$,$StoppingCriteria$,$MaxDepth$) are
            hyperparameters that denote the maximum number of splits before the tree stops
            growing, the error threshold at which the tree stops growing, and the maximum
            depth that a tree can grow respectively.
   Output: Predicted Class Labels of Testing Data ($TestPredictions$)
2  First, determine the number of unique classes ($K$) using the Training Labels and initialize a
    Tree Data Structure with a RootNode.
3  while SplitCount $<$ MaxSplits and BestGiniCost $\geq$ StoppingCriteria do
4      for TerminalNodes of depth $<$ MaxDepth do
5          Find the best split ($j^*$,$s^*$) for each TerminalNode using equations 9-11
6          Calculate the difference between Node Impurity of the Parent Node and the average
            of the impurities of the children node evaluated using the best split ($j^*$,$s^*$).
7      end
8      Split the TerminalNode with the largest difference in GiniIndex between ParentNode
        and ChildNodes.
9      The ChildNodes are new Terminal Nodes and the ParentNode is removed from the
        Terminal Nodes list.
10     Increment the SplitCount variable by 1.
11 end
12 The decision tree should now be fully generated according to the hyperparameters.
13 Next, evaluate the testing data using the learned model.
14 for testing point i in M2 testing points do
15     First, set the CurrentNode to be the RootNode.
16     while CurrentNode is not a Terminal Node do
17         if $x_i$(feature j) $\geq$ CurrentNode.threshold then
18             CurrentNode = child node to the right
19         else
20             CurrentNode = child node to the left
21         end
22     end
23     Now that we have iteratively traversed the tree to a terminal node we predict each
        testing point i as the class most associated with the terminal node.
24 end
25 Finally, return the vector of M2 predictions of the testing data as your output.
```

References

[DT05] N. Dalal and B. Triggs. Histogram of oriented gradients for human detection. *Conference on Computer Vision and Pattern Recognition*, 2005.

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2 edition, 2009.

[TK09] S. Theodoridis and K. Koutroumbas. *Pattern Recognition.* Associated Press, 4 edition, 2009.