Comparing five common classification algorithms on facial and numeral images

Christian M. Lillelund, *201408354@post.au.dk, School of Engineering, Aarhus University*

***Abstract*—X**

***Index Terms*—IEEE, IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

For humans, it doesn't take much effort to tell the difference between a picture of a dog or a cat. A natural number or a letter. A happy person or a sad person. For computers, these sort of problems are notorious hard to solve and often require many computational resources. Machine learning and computer vision deals with these issues as they encompass a range of algorithms and classification techniques to produce a model or scheme that can tell images apart and recognize similarities. In this report we study the recognition and classification of a data set containing human faces and one featuring hand-written numbers by implementing and testing five commonly used techniques (Nearest class centroid classifier, nearest sub-class centroid, nearest neighbor and two perceptron variants) using MATLAB. We split the data in a training and a test set, then train our respective model or classifier and evaluate their ability to classify correctly on the testing set. A version using all the dimensions of the data and a principal component version will be applied. We use some visualization techniques to better communicate the data representation and tables to compare them. We start by going over the basic theory behind the classification schemes, then look at the data, briefly go over implementation details and then turn to results. At the end we review and argue which scheme would make the most sense to use with these two classification problems.

Initially, we split the ORL image data set, containing 400 40x30 pixel faces of 40 different persons, and the MNIST set containing 70.000 28x28 pixel hand-written numbers 0-9. The data is vectorized, so we use built-in MATLAB functions to construct the original pictures as such:

Fig. 1. Ten random images of faces from the ORL set.

Manuscript received April 19, 2005; revised August 26, 2015.

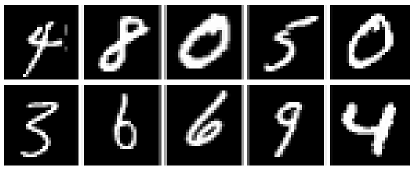

Fig. 2. Ten random images of numbers from the MNIST set.

II. Theory

This section will explain fundamental theory behind the dimensional reduction technique (PCA) and five schemes used in this report.

A. Principal Component Analysis

PCA is a procedure that transform a set of observations or samples in dimension D to a lower dimension $D-n$ while still preserving a smaller number of variables explaining the main features $X_1, X_2, ..., X_p$ in the original set. This is particularly useful when dealing with high dimension data, as this can be computationally hard and challenging to visualize. With PCA, we compute principal components d of n original samples with p features, where d is the desired output dimensionality and each dimension is a linear combination of the p features. In practice, we find eigenvectors of the covariance matrix of the original data set, sort them by highest eigenvalue score and use these as weights W in computing a linear transformation

$$y_i = W^T * x_i, i = 1, ..., N$$

. The scattering of the transformed data is the scatter matrix, a function of X:

$$S_T = \sum_{i=1}^{N} [W^T(x_i - \mu)][W^T(x_i - \mu)]^T$$

. The weights W can be found by applying eigenanalysis and taking the eigenvectors with the highest score, more formally optimizing:

$$W* = \arg\max_c Tr(W^T S_T)$$

subject to $W^T W = I$. We end with a data set containing fewer ($d < D$) dimensions.

B. Nearest class centroid classifier

The NCC classifier assigns labels l_i to N observations determined by which class c_k's mean (centroid) the observation

x_i is closest to. We make the assumption that each class follow a normal distribution, as they are given equal importance in the classification algorithm. The mean class vector is given by:

$$\mu_k = \frac{1}{N_k} \sum_{i,li=k} x_i, k = 1, ...K.$$

To classify any observation x_i we find the smallest distance to any mean vector and assign the label of that vector to it, more formally:

$$d(x_i, \mu_k) = ||x_i - \mu_k||_2^2$$

C. Nearest subclass centroid classifier

Similar to nearest class centroid, but each class c_k now has subclasses m that follow a normal distribution and has a mean vector μ_{km}:

$$\mu_{km} = \frac{1}{N_{km}} \sum_{i,li=k,qi=m} x_i$$

where N_{km} denotes the number of observations in a given subclass and x_i is a observation with a subclass label qi. Like NCC, the distance to the subclass mean is used to classify each observation:

$$d(x_i, \mu_{km}) = ||x_i - \mu_{km}||_2^2$$

The number of subclasses is a hyper parameter for NSC and must be decided prior. Nearest subclass classifier is a compromise between nearest mean and nearest neighbor and combines the flexibility of nearest neighbor with the robustness of nearest mean, which we describer next. When the number of subclasses m is equal to N samples, we have the nearest neighbor classifier.

D. Nearest neighbor classifier

The nearest neighbor (NN) classifier is a simple algorithm where each sample is assigned to the class of its closest neighbor, or the most common class among its k nearest neighbors in the k-NN variant. Pure NN is when $k = 1$, but often $k > 1$ where a majority vote takes place. The algorithm uses euclidean distance between a test sample x_i with class c_k and a training one y_i:

$$d(x_i, y_i) = ||x_i - y_i||_2^2$$

E. Perceptron learning with backpropagation

A perceptron is a supervised algorithm of binary classifiers, that classify whether or not an input sample x_i belong to class c_k. It uses the linear discriminant function $f(x) = w^T * x + w_0$, where w are weights, $x \varepsilon\ \mathbb{R}^{\mathbb{D}}$ a feature vector and w_0 the bias. w represent the orientation of the discriminant hyperplane, and w_0 a offset from the origin. Given weights w, sample x_i and classes c_1 and c_2, the function splits the feature space into these two classes. If $g(w, x_i) > 0$, x_i belongs to c_1 and if $g(w, x_i) < 0$, x_i belongs to c_1. If $g(w, x_i) = 0$, it can be classified to either class. The resulting scalar is the distance of x_i to the hyperplane. The function is used in our perceptron to follow, where we define a binary label l_i for each sample to express the criterion function:

$$f(w^*, x_i) = l_i g(w, x_i) = l_i w^{*T} x_i \geq 0, i = 1, ..., N$$

All samples are correctly classified, if $f(w_*, x_i) \geq 0$. To produce such result, we need to optimize weights w_* by the perceptron criterion function for w_*, where $\jmath_p(w_*) = 0$ would be a solution for w_*:

$$\jmath_p(w) = \sum_{x_i \varepsilon \chi} -f(w, x_i) = \sum_{x_i \varepsilon \chi} -l_i w^T x_i$$

To optimize $\jmath_p(w)$, we use the gradient at w and follow it to update w, where η is the rate of change (learning rate) and expresses how fast it converges, and ε express a vector set of mislabeled samples:

$$w(t+1) = w(t) - \eta(t) \nabla \jmath_p = w(t) + \eta(t) \sum_{x_i \varepsilon \chi} l_i x_i$$

At each learning iteration, the weights $w(t)$ are "punished" by the sum of misclassified samples scaled, leading to convergence as the result of criterion function $\jmath_p$ will be higher. This is known as backpropagation. The algorithm to come will elaborate on this.

F. Perceptron learning with MSE

As an alternative to backpropagation, one can apply least-mean square regression to obtain the perceptron weights W. Here we use a criterion function $\jmath_{LSE}$ that involves all of the samples, not just the misclassified ones. Previously we made the inner products of $f(X) = W^T * X$ positive, now we look to find a solution to $W^T X = t_i, i = 1, ..., N$, where t_i are arbitrarily set positive constants. This is written as $X^T * w = b$ in matrix form. If X is nonsingular, we could write $W = X^-1b$ and have a formal solution, but this is often not the case, however by obtaining the pseudo-inverse of X one can find a solution for the weights:

$$X^{\dagger} = (X^t X)^{-1} X^T$$

Examining the criterion function, we look to minimize:

$$\jmath_{LSE} = ||W^T X - B||_F^2$$

By setting the derivative of this to zero with respect to W, we get:

$$\nabla \jmath_{LSE} = 0 => 2XX^W = 2XB^T$$

Hence:

$$W = (XX^T)^{-1} XB^T = W^{\dagger} B^T$$

If W is square and nonsingular, the pseudo-inverse is simply the regular inverse. Also it holds at $X^{\dagger} X = I$, but $XX^{\dagger} \neq I$, though a MSE always exists, hence $W = X^{\dagger} b$ is an MSE solution to $X^{\dagger} w = b$.

III. Implementation

The implementation uses MATLAB to design and run the different schemes on the training data and evaluating on the test data. For understanding purposes, this section will briefly lay out the algorithms described in theory as pseudo-code how we implemented them.

A. PCA

The algorithm centers the data, then simply calculates eigenvectors from the variance matrix, sort them after highest eigenvalue and uses the n vector as principal components in representing the data. Listing 1 shows this.

```
pca_reduce(trainData, nDimensions) {
  trainData = trainData − mean(trainData) for
      rows in trainData
  eigVectors, eigValues = eig(cov(trainData))
  eigVectors = sort(eigVectors, 'descending')
  pc = eigVectors(from 1 to nDimensions)*
      trainData
}
```

Listing 1. Implementation of PCA.

B. NCC

Simply calculate a mean vector for c_k classes by summing the samples that belong te class and dividing that with the number of N samples in a given class, as expressed in listing 2.

```
train_ncc(trainData, trainLbls, nClasses) {
  mu = zero vector by size of trainData*
      nClasses
  n = zero vector by size of 1*nClasses
  index = 1;

  for (i = 1; i <= nClasses;i++) {
    n(i) = sum(num of classes i in trainLbls)
    mu(i) = sum(each traning sample from index
        to n(i))/n(i)
    index = index+n(i)
  }
}
```

Listing 2. Implementation of NCC.

C. NSC

We make an initial k-means clustering of the training data for $k = m$ subclasses as hyper parameter and use the resulting centroids to construct a new centroid matrix for each subclass m spanning c_k*m. Listing 3 shows this.

```
train_nsc(trainData, trainLbls, nClasses,
    nSubClasses) {
  centroids = zero vector by size of trainData
      *(nClasses*nSubClasses)
  n = zero vector by size of 1*nClasses
  index = 1;
  centroidIdx = vector to keep idx of centroids

  for (i = 1; i <= nClasses;i++) {
    n(i) = sum(num of classes i in trainLbls)
    [idx,clst] = kmeans(trainData for index+n(i
        ), nSubClasses)
    centroids(for idx in centroidIdx) = clst
    index = index+n(i)−1
  }
}
```

Listing 3. Implementation of NSC.

D. Perceptron with backpropagation