Comparing five common classification algorithms on the ORL and MNIST data set

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

***Abstract*—Today's computer systems offer a significant potential for training and testing machine learning techniques and classification algorithms, however the choice of algorithm depend on the problem at hand, one's amount of train and test data and the computational resources at one's disposal. In this report we review one dimensionality reduction technique and five common classification schemes and evaluate their performance on the ORL (human faces) and MNIST (hand-written numbers) data set. We find that simple algorithms like the Nearest Neighbor classifier work great performance and accuracy wise when training on the small data set, ORL, but struggle to keep up with more complex algorithms like a Perceptron using backpropagation on the large set, MNIST, as the dimensionality and amount of samples is significantly increased.**

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

I. Introduction

For humans, it does not 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[1]. 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 classifier and evaluate its ability to classify correctly on the testing set. A version using all the dimensions of the training data samples and a principal component version will be applied. We use some MATLAB 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 from zero to nine. The data is vectorized, so we use built-in MATLAB functions to construct the original pictures as such:

Figure 1: *Ten random images of faces from the ORL set.*

Figure 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 the 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[2]. 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 given by [3]:

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

Report revised December 10, 2018.

[1][Terence Mills(2018)]

[2][Casella et al.()Casella, Fienberg, and Olkin]

[3][Iosifidis(2017)]

. The weights W can be found by applying eigenanalysis and taking the eigenvectors with the highest score, more formally optimizing [4]:

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

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 [5]:

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

To classify any observation x_i we find the smallest distance to any mean vector and assign the label of that vector to x_i, 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 [6]:

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

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 [7]. 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:

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

[4][Iosifidis(2017)]
[5][Iosifidis(2017)]
[6][Iosifidis(2017)]
[7][Veenman and Reinders(2005)]

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 a 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[8]. 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[9] 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[10]:

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

Examining the criterion function, we look to minimize:

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

[8][Iosifidis(2017)]
[9][Iosifidis(2017)]
[10][Duda et al.(2012)Duda, Hart, and Stork]

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

$$\nabla_{J_{LSE}} = 0 => 2XX^W = 2XB^T$$

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 and test data. For understanding purposes, this section will briefly lay out the algorithms described in theory as pseudo-code.

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 a principal component in representing the data. Listing 1 shows the PCA as pseudo-code.

```
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

NCC simply calculates a mean vector for c_k classes by summing the samples that belong to the 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

NSC makes 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. NNC

The algorithm takes as input the training and test data, as well as training labels. It then iterates through every test sample x_i and compares to every single training sample X_i. It classifies x_i to the class of X_i it is closest too in Euclidean distance. Listing 4 shows this.

```
train_nnc(trainData, trainLbls, testData) {
  distances = zero vector by size testData*
      trainData
  resultLabels = zero vector by size testData*1
  nTestImages = num of test images
  nTrainImages = num of train images

  for (i = 1; i <= nTestImages; i++) {
    for (j = 1; j <= nTrainImages; j++) {
      distances(i,j) = norm(testData(i)-
          trainData(j), 2)^2.
    }
    trainIndex = min(distances(i))
    resultLabels = trainLbls(trainIndex)
  }
```

Listing 4: *Implementation of NNC.*

E. Perceptron with backpropagation

The algorithm trains a perceptron to correctly classify each sample x_i to class c_k. We start with an empty vector χ, then create a label l_i for sample x_i setting it to 1 if it corresponds to our current class, otherwise -1. We then use a one-vs-rest criterion function $f(w, x)$ for the x_i's, save the wrongly classified samples and labels, then summing the wrong samples times their label and use this value to update weights w_k iteratively. The learning eta η is a hyper parameter. Listing 5 shows the pseudo-code for this.

```
train_perp_bp(trainData, trainLbls, eta, nClasses)
    {
w = random W matrix with size trainData*nCLasses
maxIters = max itertions
for (k = 1; k <= nClasses; k++) {
  wrongLabels = empty matrix to save labels
  done = false, i = 0
  while (not done and nIters < maxIters) {
    X = empty matrix for wrong x_i's
    for (i = 1; i <= nTrainImages, i++) {
      x_i = trainData(i)
```

```
        if(trainlbls(i) = k) label = 1 else label =
            -1
        if (label*w(k)*x_i) < 0
          save x_i to X, save label to WrongLabels
      }
      sumOfWrongs = sum(WrongLabels * X)
      w(k) = w(k) + eta * sumOfWrongs
      ifEmpty(X)
        done = true
      }
    }
  }
```

Listing 5: *Implementation of a perceptron with backprogapation.*

F. Perceptron with MSE

The algorithm for MSE is simplified, but possibly more computational demanding. We start by finding the pseudo-inverse of our training data set, then create a label vector for each class c_k with $l_i = 1$ if the label is in the class or $l_i = -1$ if it is not. Finally we find the MSE solution by multiplying the two for weight W_k. Listing 6 shows this.

```
train_perp_mse(trainData, trainLbls, nClasses) {
  w = random W matrix with size trainData*nCLasses
  X = pseudo-inverse of trainData

  for (k = 1; k <= nClasses; k++) {
    Labels = empty matrix to save labels
    for (i = 1; i <= nTrainImages; i++)
      if (trainlbls(i) = k)
        save 1 to Labels
      else save -1 to Labels
    }
    w(:,k) = X*Labels;
  }
}
```

Listing 6: *Implementation of NSC.*

IV. Results

The five classification schemes are trained and tested on the ORL and MNIST data set with scatter plots to visualize the resulting labels compared to the testing ones and a confusion matrix that depicts the predicated and true classes. Each algorithm is tested with both the PCA and the full version of the data. All results were done with MATLAB R2018a on an Intel i5 4690K @ 3.50Ghz CPU. We start by showing the output of the PCA analysis, and then proceed to the classifications.

A. PCA

PCA was performed on both ORL and MNIST data set for number of dimensions $D = 2$ with MATLAB. Table I details the computation time for PCA and figure 3 and 4 shows a scatter plot with two principal components, i.e. the eigenvectors with highest eigenvalues.

Data set	Method	Execution time in s
ORL	PCA	0.33
MNIST	PCA	0.83

Table I: *PCA on ORL and MNIST data set.*

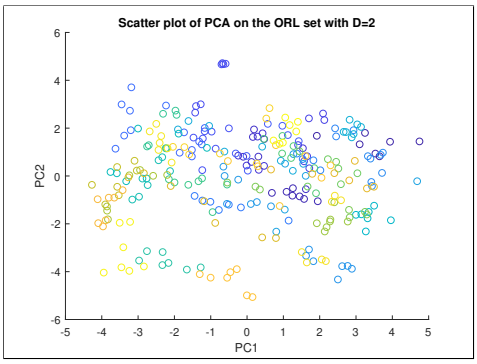

Figure 3: *Plot of two PC's on ORL. Data labels are colors.*

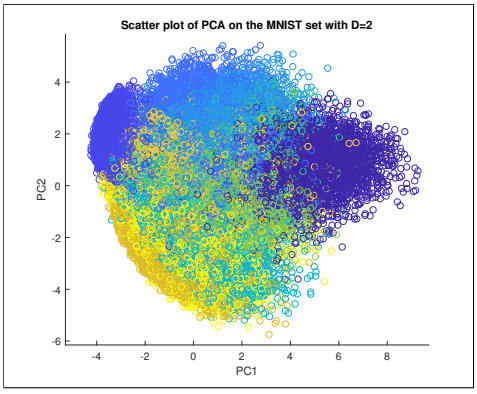

Figure 4: *Plot of two PC's on MNIST. Data labels are colors.*

B. NCC

Nearest Centroid was conducted on ORL and MNIST. To test the algorithm, we find the class c_k's mean to which a sample x_i is closest to by calculating the Euclidean distance (L2-norm) between a test sample and the mean vector. Table II details performance and accuracy, while figure 5 and 6 shows the resulting labels on scatter plots. The tests were performed on full-dimensional sets and PCA reduced sets. Accuracy is the number of resulting labels that equal the testing labels divided by the number of test samples.

Data set	Accuracy in %	Execution time in s
ORL	94,1%	0.00
ORL w. PCA	15,8%	0.00
MNIST	75,2%	0.34
MNIST w. PCA	14,9%	0.00

Table II: *Results for NCC performed on ORL and MNIST.*

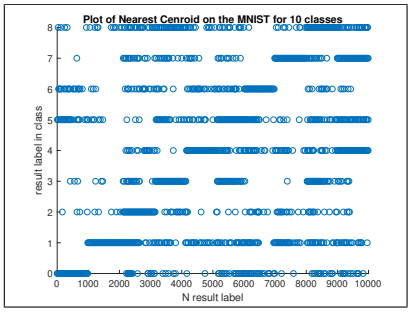

Figure 5: *Plot results of NCC on ORL.*

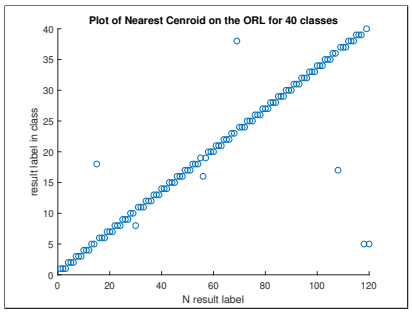

Figure 6: *Plot results of NCC on MNIST.*

C. NSC

Nearest Subclass Centroid was performed on ORL and MNIST with hyper-parameter $N_{sc} = 2, N_{sc} = 3, N_{sc} = 5$ for number of subclasses. Our algorithm returns centroids for number of classes $N_c = N_{ck} * N_{sc}$. We classify a testing sample to a class of which its centroid it is closest to. Table III details the result. N_{sc} is number of subclasses. Figure 7 and 8 show the NSC on the ORL and MNIST data set. The blue color represents test labels, the red color represents result labels. Each test sample is plotted to their class.

Data set	N_{sc}	Accuracy in %	Execution time in s
ORL	2	93.3	0.15
ORL	3	95	0.15
ORL	5	95.8	0.15
ORL w. PCA	2	9.1	0.13
MNIST	2	86.1	4.07
MNIST	3	88	5.99
MNIST	5	90	9.21
MNIST w. PCA	2	14.3	0.14

Table III: *Results for NSC performed on ORL and MNIST.*

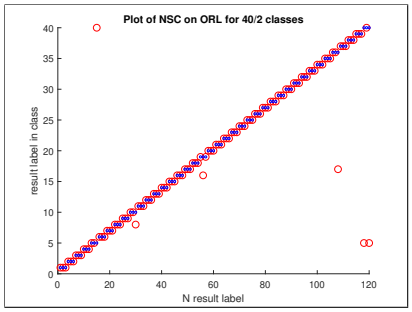

Figure 7: *Plot results of NSC on ORL with* $N_{sc} = 2$.

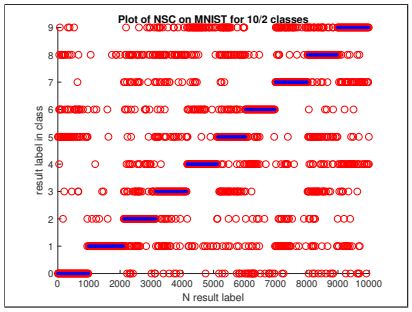

Figure 8: *Plot results of NSC on MNIST with* $N_{sc} = 2$.

D. NNC

Nearest Neighbor classifier was performed on ORL and MNIST. To classify a test sample, the algorithm calculates the Euclidean distance to every training sample and assigns the test to whichever training sample it is closest to. Table IV shows the results with accuracy and execution times. Table V is a confusion matrix to visualize the classification result. It shows samples known to be in class c_k classified correctly to c_k, i.e. 973 samples were rightly classified to class 1.

Data set	Accuracy in %	Execution time in s
ORL	95.8%	0.04
ORL w. PCA	11.6%	0.01
MNIST	96.9%	676.07
MNIST w. PCA	14.9%	264.65

Table IV: *Results of NNC performed on ORL and MNIST.*

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	973	1	1	0	0	1	3	1	0	0
2	0	1129	3	0	1	1	1	0	0	0
3	7	6	992	5	1	0	2	16	3	0
4	0	1	2	970	1	19	0	7	7	3
5	0	7	0	0	944	0	3	5	1	22
6	1	1	0	12	2	860	5	1	6	4
7	4	2	0	0	3	5	944	0	0	0
8	0	14	6	2	4	0	0	992	0	10
9	6	1	3	14	5	13	5	4	920	5
10	2	5	1	6	10	5	1	11	1	967

Table V: *Confusion matrix for a NNC model tested on MNIST.*

E. Perceptron with BP

A perceptron was trained on both data sets with the learning rate η as hyper-parameter for $\eta = 0.01$, $\eta = 0.1$, $\eta = 1$ and $\eta = 10$. For each test sample, we calculate the decision function $y = w(k) * x_i$ for each class c_k and classify it to the class with maximizes the function. A maximum number of iterations per class is set to 100. Table VI details the results, figure 9 and 10 show plots of the resulting labels and 11 and 12 show the classification perceptron line in 1D and 2D with PCA.

Data set	η	Accuracy in %	Execution time in s
ORL	0.01	86.6	1.39
ORL	0.1	84.1	1.30
ORL	1	82.5	1.36
ORL	10	84.1	1.34
ORL w. PCA	0.1	4.1	1.13
MNIST	0.01	87.3	131.39
MNIST	0.1	86.6	115.50
MNIST	1	86.2	115.60
MNIST	10	79.3	113.25
MNIST w. PCA	0.1	7.4	1.16

Table VI: *Results for a perceptron with BP performed on ORL and MNIST.*

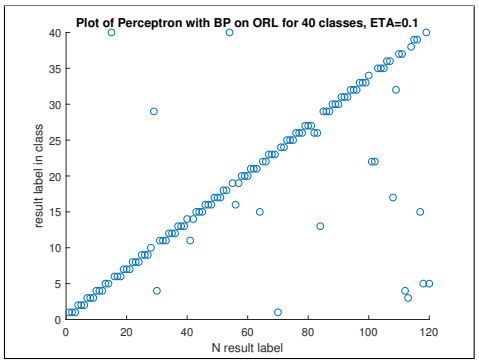

Figure 9: *Plot results of training a perceptron with BP on ORL with* $\eta = 0.1$.

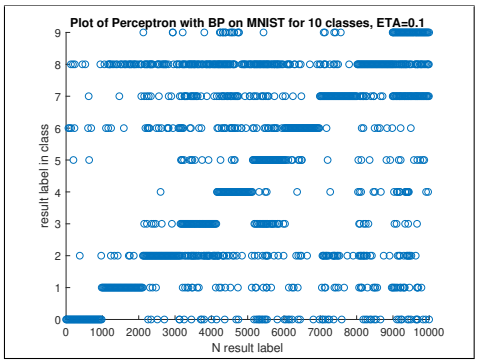

Figure 10: *Plot results of training a perceptron with BP on MNIST with* $\eta = 1$.

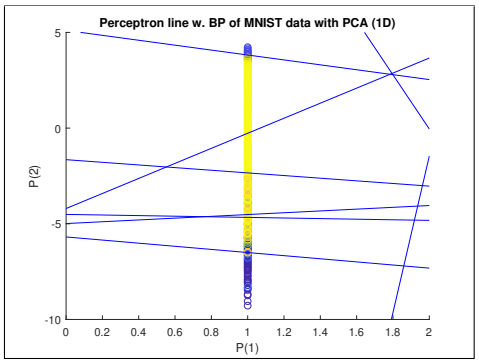

Figure 11: *Plot lines of perceptron with BP for classes on MNIST in 1D.*

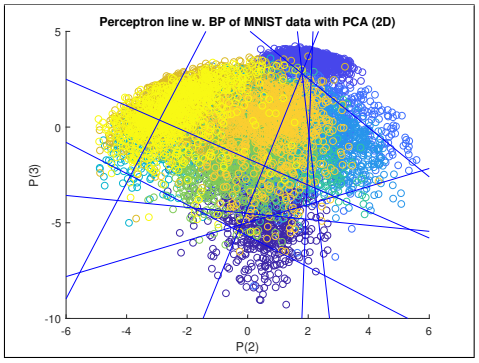

Figure 12: *Plot lines of perceptron with BP for classes on MNIST in 2D.*

F. Perceptron with MSE

A perceptron using least-squares was trained on both data sets. Like with backpropagation, we find the class that maximize the decision function $y = w(k) * x_i$ for each sample and classify it to that class. Table VII details the results, figure 13 and 14 shows plots with the resulting labels.

Data set	Accuracy in %	Execution time in s
ORL	90%	0.05
ORL w. PCA	10%	0.01
MNIST	86%	6.79
MNIST w. PCA	18.4%	0.53

Table VII: *Results for a perceptron with least-squares (MSE) performed on ORL and MNIST.*

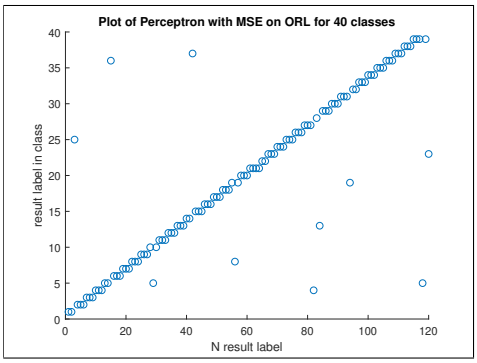

Figure 13: *Plot results of training a perceptron with least-squares (MSE) on ORL.*

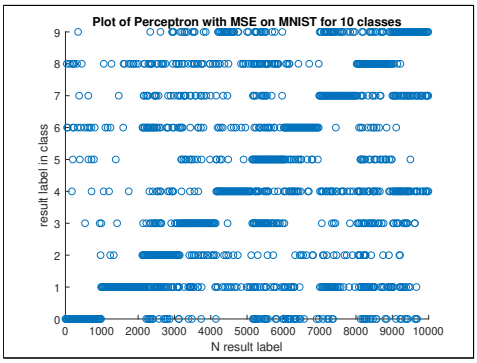

Figure 14: *Plot results of training a perceptron with least-squares (MSE) on MNIST.*

V. Discussion

As expected, the algorithms performed differently given the complexity and dimensionality of the data set. If a classification scheme is fast and accurate on a small data set, like ORL ($D = 1200$), it might not replicate on a larger set like MNIST ($D = 60000$). PCA showed a computational easy way to reduce dimensionality, but accuracy with PCA data lacked in all our tests, averaging around 15%, even though the execution times were similar to that of the full dimensional sets. Nearest Centroid Classifier (NSC) gained an impressive accuracy on ORL (94.1%), but struggled on MNIST (75.2%). Nearest Subclass Centroid (NSC) proved high classification precision, with a 2% difference between 2 and 5 subclasses on ORL, and

3.9% on MNIST, however execution times doubled on MNIST with five subclasses compared to just two. Nearest Neighbor (NNC) gave us the best results in terms of accuracy (95.8% and 96.9%), but suffered from very high execution times. Our perceptron with BP showed that a lower learning rate $\eta < 0.01$ gave better accuracy with similar execution times as tests with higher learning rate $\eta > 1$. The perceptron performed faster than nearest neighbor (around 2 and 11 minutes on MNIST, respectively) with similar accuracy, showing that a perceptron should be preferred over NNC on larger data sets. The one using least-squares (MSE) gave a sound accuracy on both sets, lower execution times and is a much more simple algorithm to implement and comprehend, though solving the generalized inverse often requires plenty computational resources.

VI. Conclusion

In this report we have explained the workings of five common classification algorithms including principal component analysis (PCA), the mathematics behind them and benchmarked them on two famous data sets, the ORL set containing 40 different types of facial images and the MNIST set featuring images of numbers from zero to nine. We found that simple algorithms like nearest neighbor classifier (NNC) performs excellent on simple data sets with low dimensionality (like ORL), but struggles on more complex sets like MNIST. In such case, we found that a perceptron trained either by backpropagation or MSE is more suited because it is computational more cost effective, given a boundary is set on number of convergence attempts.

References

[Casella et al.()Casella, Fienberg, and Olkin] G Casella, S Fienberg, and I Olkin. *Introduction to statistical learninglinear algebra with applications*. ISBN 9780387781884. doi: 10.1016/j.peva.2007.06.006.

[Duda et al.(2012)Duda, Hart, and Stork] R O Duda, P E Hart, and D G Stork. *Pattern Classification*. 2012.

[Iosifidis(2017)] Alexandros Iosifidis. Introduction to Machine Learning. *Introduction to Machine Learning*, apr 2017.

[Terence Mills(2018)] Terence Mills. Machine Learning Vs. Artificial Intelligence: How Are They Different?, 2018. URL https://www.forbes.com/sites/forbestechcouncil/2018/07/11/machine-learning-vs-artificial-intelligence-how-are-they-different/.

[Veenman and Reinders(2005)] Cor J Veenman and Marcel J T Reinders. The nearest sub-class classifier_a compromise between the nearest mean and nearest neighbor classifier.pdf. 27(9):1417–1429, 2005.