AMATH 582 HW 4: SVM, LDA, And Decision Trees on MNIST

Hongda Li

March 9, 2021

Abstract

The objective is to develope basic practical and theoretical understandings of some classical machine learning methods. We use the MNIST data set and the models are Linear Discriminant Analysis, Principal Component Analysis, Support Vector Machine, and Binary Decision Tree. Matlab is the major tool. PCA is used as a dimensionality reduction technique for the data. Models are trained on the projection onto the principal component mode to increase efficiency. Models are trained to distinguished pair of of digits, or all classifier all of the digits. Metrics used are the errors between the predicted labels and the actual labels and the confusion matrix. The metrics is measure on training, validations, and test set. Theory and intuitions behind the algorithms and practical details are explored in the paper.

1 Introduction and Overview

We were given the Mnist data set, which are images of hand written digits, packed into tensor of size $28 \times 28 \times N$. 60k of them are used for training, and 10k of them are used for testing. Samples for each digits are approximately homogenous.

The classification that we are interested in are SVM: Support Vector Machine, LDA: Linear Discriminant Analysis, and Decision Tree (DTree).

Principal Component Analysis (PCA) is used as a dimensionality reduction technique for the data set. More information on the theory part.

Confusion matrix is used as a performance metrics for all above classifications. And this metrics is being measured for training, validations, and the test set. It's used because it exposes more information than the errors.

In addition, visualization will be made for PCA to demonstrate the low rank property of the data set.

2 Theoretical Background

Most of the ideas discussed below are referencing sources from Mathworks, Sklearn, lecture notes of Math 253 from San Jose State University, The *Book Elements of Statistical Learning* from Springer by Trevor Hastie et al, and the book *Data-Drive Modeling & Scientific Computation* from the University of Washington by Nathan Kutz.

2.1 Linear Discriminant Analysis

Linear discriminant analysis comes in many different types of variance, and it can be regularized as well. The most simple is: LDA, which seeks to look for a linear subspace such that the projection of the labeled data onto the line has maximal distance from the global central, and minimal variance within the clusters. The model assume homogenous Gaussian distribution of the data points. If Non-homogenous Gaussian clusters are involved, quadratic discriminant analysis is needed for the classifier. However if QDA classifier is used, then the covariance matrix for the features for each cluster will have to be diagonal, implying the fact that projecting onto the PCA modes before the analysis will be helpful for QDA.

WLOG, let's assume a binary classification problems with 2 classes: C_1, C_2. The features of each sample is a vector denoted by x_i and the subspace we project them onto is a vector denoted as v, then, the centroid for one of the cluster projected onto the line can be expressed as[1]:

$$\mu_1 = \frac{1}{|C_1|}\sum_{x_i \in C_1}\left(v^T x_i\right) = \frac{v^T}{|C_1|}\sum_{x_i \in C_1}(x_i) \tag{1}$$

And conveniently, the centroid of the cluster can be expressed as:

$$m_1 = \frac{1}{|C_1|}\sum_{x_i \in C_1}(x_i) \tag{2}$$

Consider the distance between μ_1, μ_2 on the projected space, which can be expressed as:

$$\begin{aligned}(\mu_1 - \mu_2)^2 &= (v^T m_1 - v^T m_2)^2 = (v^T(m_1 - m_2))^2 \qquad (3)\\ (v^T(m_1 - m_2))^2 &= v^T(m_1 - m_2)\cdot(m_1 - m_2)^T v\\ S_{\text{between}} &= (m_1 - m_2)(m_1 - m_2)^T \in \mathbb{R}^{d\times d}\end{aligned}$$

Where we have define the rank-one matrix that measures the distance between the projected mean of the clusters.

Now consider another matrix which will be helpful for us: WLOG, the variance for the class C_1 can be computed by:

$$\begin{aligned}s_1^2 &= \sum_{x_i\in C_1}\left((v^T x_i - v^T m_2)^2\right) = \sum_{x_i\in C_1}\left(v^T(x_i - m_1)(x_i - m_1)^T v\right)v^T \qquad (4)\\ s_1^2 &= \left[\sum_{x_i\in C_1}((x_i - m_1)(x_i - m_2))\right]v = v^T S_{\text{within}} v\end{aligned}$$

Then, the problem of maximizing the centroid of the cluster onto the sub-space and minimizing all the variance of the cluster can be summarized as:

$$\max_{\|v\|=1}\left\{\frac{(\mu_1-\mu_2)^2}{s_1^2+s_2^2}\right\} = \max_{\|v\|=1}\left\{\frac{v^T S_{\text{between}} v}{v^T S_{\text{within}} v}\right\} \tag{5}$$

And computationally, this become an augmented Rayleigh Quotient, which can be solved by looking for vector in the Eigenspace of the matrix $S_{\text{between}} S^{-1}_{\text{within}}$.

Take notice that, this formation is from an optimization point of view instead of using Bayes Theorem, and using the idea of Bayes theorem, one can formulate the problem for Quadratic Discriminant Analysis and derive the computational problem.

Take note that, for multi-class, LDA take the squared distance between each cluster from the global centroid of all the data point as a measure for the spread of the clusters.

In addition, we need to notice that LDA can be used both as a dimensionality reduction technique and a classifier. The Eigenspace of the matrix $S_{\text{between}} S^{-1}_{\text{within}}$ is subspace which, maximal separation between the data occurred. However, it doesn't have to be an orthogonal subspace.

2.2 Principal Component Analysis

Principal Component Analysis is used as a dimensionality reduction technique[2]. Consider the Column Data Matrix X and its SVD decomposition of the matrix:

$$X = U\Sigma V^T \implies U^T X = \Sigma V^T \tag{6}$$

Where: U, V are both orthogonal matrices. Assuming that the matrix X is $m \times n$ and in the case of our data set, $m << n$, then the matrix $U \in \mathbb{R}^{m\times m}$, $\Sigma \in \mathbb{R}^{m\times m}$, and $V^T \in \mathbb{R}^{m\times n}$.

The Interpretation of SVD is simple. We decompose the column data matrix into Principal Components that explains the most variance (Columns of the matrix U) and the singular value, which explain the importance of each mode (Diagonals of matrix Σ), and the profile vector, which describes the composition for each of the sample (Column vector of the matrix V^T, rows of the matrix V).

Conveniently, the principal components of the data matrix is ordered with the singular value and singular values are ordered in descending order.

This means that, we can truncate the rows of the matrix ΣV^T yet still retaining most of the information for each column of the data matrix. This is the "Eckart-Young" Theorem, a major idea behind PCA and dimensionality reduction. Assuming that we now truncate the SVD into using matrices of the size $m \times k, k \times k, k \times n$ for U, Σ, V^T where $k << m$ then it also has the added benefit that the sample spans all the subspace $\mathbb{R}^k$, implying that the covariance matrices for the classes are going to be non-singular. Which is a requirement for Quadratic Discriminant Analysis.

2.3 Support Vector Machine

The idea of SVM is to find the hyper plane between the clusters such that it maximizes the margin between the boundary of the 2 clusters. This is intrinsically a binary classification method[4].

Computationally the problem is phrased as a Quadratic Programming problem. The SVM is computationally phrased as[6]:

$$\min_{\beta,\beta_0} \|\beta\| \text{ s.t: } \begin{cases} y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i & \forall i \\ \xi_i \geq 0, \sum_i (\xi_i) \leq \text{constant} \end{cases} \tag{7}$$

Where $y_i \in \{1, -1\}$ are the binary label of the sample, and ξ_i is the penalty for cross the boundary for each of the support vector, and β is the vector the defines the hyperplane.

In practice, to get the best separations for different topology of the cluster, kernel function is often used as a way of mapping the same data from a lower dimension to a high dimension, while the theoretical analysis of the dual problem allows improve the speed and make use of only the pair-wise inner product under the new basis[7].

2.4 Binary Decision Tree

The idea of binary tree is a simple yet powerful concepts. A tree is created to partition the feature space on each node of the tree. Each branching of the tree assert one conditions on one of the attribute, (e.g: age older than 18), and each leaf node consists of samples with almost the same labels.

There are many variance of the decision tree, but for our purposes, we will be focusing on the default that comes with the "fictree" command in matlab[3].

Decision tree unlike method described above, doesn't have smooth decision boundary, making it sensitive to noise and extrapolation of the data. However, it's it has a variety of hyper parameters available for tunning and it's a interpretable[5].

3 Algorithm Implementation and Development

The challenging part of machine learning practice in this case is data preprocessing and dimensionality reduction of the data. Hence for this part we will be highlighting the procedures for PCA, unit variance zero mean Standardization, and MinMax Standardization, and the generic training process of various models.

Others parts of the practical implementations that are not listed here involves the procedures of visualizing the results, setting up the parameters, and filtering out certain digits from the training set.

3.1 Data Standardization

There are 2 standardization techniques used in the training process of various models. The zero mean unit variance standardization(Algorithm 1, implemented in: ImageNormalize), which is used for the PCA, and the LDA. The MinMax standardization(Algorithm 2 implemented in MinMaxStd), which is used for the SVM. The SVM only works for points that have relative distance due to the penalty of the corresponding optimization problem, hence MinMax standardization is needed.

Algorithm 1: Algorithm 1: Zero Mean Unit Variance Standardization

```
Input: Data: D
D := Flattten the Tensor: D into a column data matrix, convert the data type as well.
D := D - mean(D, 1)
D := D./std(D, 1)
Return: D
```

Algorithm 2: Algorithm 2: MinMax Standardization

```
Input: Data D
Precondition: Algorithm 1 Has been run on D.
D := data - minAll(D)
D := data./maxAll(D)
Return: D
```

3.2 PCA Dimensionality Reduction

Algorithm 3: Algorithm 3: PCA Dimensionality Reduction

```
Input: Data: D EnergyLevel: E
Preconditions: The data, D is processed by Algorithm 1
Perform SVD on dat matrix D to obtain matrices: S, Sigma, V
EnergyS := cumsum(Sigma)/sumAll(Sigma)
Idx := the index of first number in EnergyS such that it's > E
ProjData := Sigma * V$^T$
ProjData := ProjData(1: Idx, :)
Projector := U(1: Idx, :)$^T$
Return: ProjData, Projector
```

Algorightm 3 describes the procedure for PCA dimensionality reduction. Variable "D" is the data matrix of the images, where each images are packed as columns of the matrix D, and "E" is the cumulative energy level we want for all the modes that we are projecting onto. The function then, returns the data described using all first "Idx" principal modes, and return the projector, which is just the matrix U^T. Note, the function "sumAll" does sum up all the elements in the tensor and it's out put is a scalar. Implemented in the class MNISTPCA.

3.3 Training Generic Models

Algorithm 4 refers to the generic procedures of producing a model given the training and test MNIST Data set. Which is important for checking the constancy of the cross validation and the training set, and it's then benchmarked on the test set to see if there are potential for over-fitting.

For our purpose, we trained SVM, LDA, and Decision Tree using the Gaussian Kernel function for SVM, linear discriminant for LDA and they are all trained on the first 66 principal mode which covers just over 50% of the cumulative energy on the singular value spectrum. The choice of using 50% of the energy is aimed for reducing noise in the data, which will help with models that are sensitive to noise such as the SVM and the decision tree. When training the SVM model, all the data point is normalized using Algorithm 2 to transform the features such that the feature lies in $[0, 1]^{66}$. The algorithm is implemented by: PrepareData and the class ParameterPack.

4 Computational Results

When reading this section, it's recommended to open two instances of this file, because all the figures are located at the end of the appendix for formatting reasons.

4.1 Visualizing Low-rank Structure Using PCA

The low-rank structure of the MNIST data is exposed by the decay of the singular value. Shown on left side of Fig 1. In addition, some form of clustering on the mode 1, 5, 8 is shown on the right side of Fig

Algorithm 4: Algorithm 4: Generic Model Training Process

```
1: Input: Training Data and Labels: TrD,TrL Testing Data and Labels: TD, TL
2: Standardize the data: TrD, TrL, TD, and TL appropriate to the model
3: X, X1, Y, Y1 := Permuate the trainig data, 10% for corss validation, 90% for training using TrD, TrL
4: Train the model using X, Y
5: Produce labels using model on X, Y, X1, Y1
6: Produce labels using model on TD, TL
7: Return: All the label produced, data set and the model itself
```

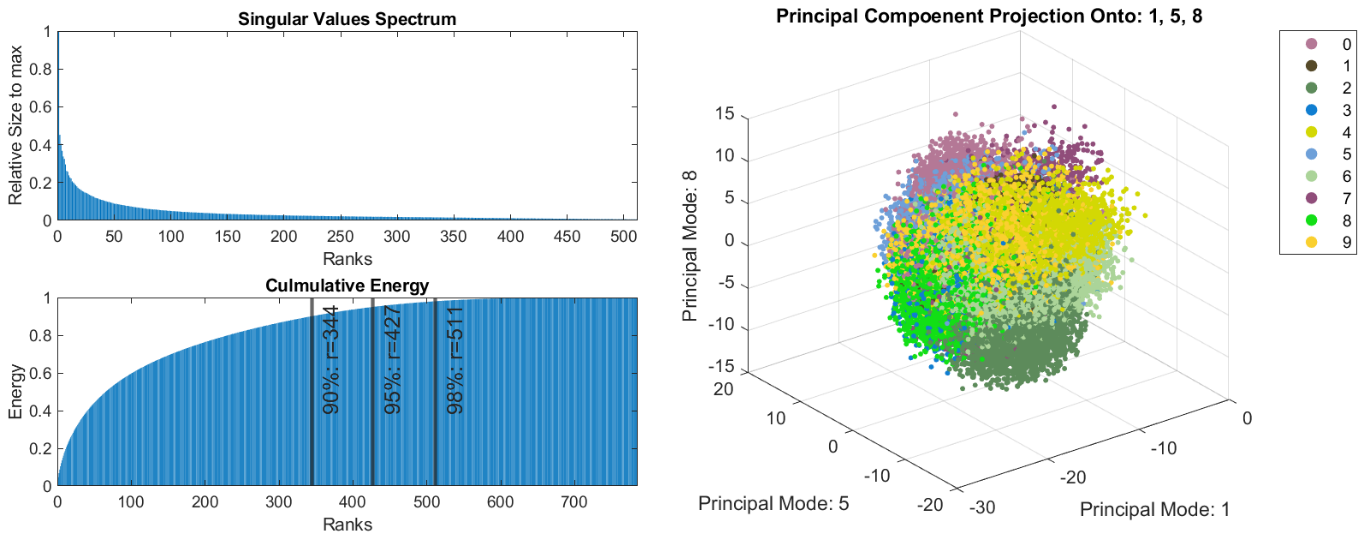

Figure 1: Figure 1: PCA Project and Singular Spectrum

1 where each color represents a digit, and is having their own approximate region in the projected space. 66 principal modes are used for training all the models model, based on the assumption that most of the variances are noises.

4.2 Performance Comparison on 2,3 Digits Separation

The strategy for looking for the hardest digits to separate is referencing the confusion matrix across the test set of the LDA method on all 10 digits, this is shown in Fig 5. This is achieved by training all the digits on the LDA model on a one vs one setting. Focusing on the misclassified rate for each pairs of digits, it's not hard to see that the digits $4, 9$ are the digits where the model make most of the mistakes. And the digits that the model makes the least amount of mistakes are the digits: $1, 0$. Shown in Fig 2 the LDA misclassified 13.4% of the instances of 9 as 4 and 1.5% instances of 4 as 9 on the test set, but sightly lower on the training set and validation set, indicating a potential for **over fitting**. Between the easiest digits: 0, 1, the LDA models misclassified 0.2% of the digits 0 as 1 and 0.1% of the digits 1 as 0 on the test set (Fig 3). Training error and the test error is extremely close, indicating a very good separations of the distribution between these 2 classes on the PCA modes. LDA shows some efforts when it tries to separate 3 digits, and the confusion matrix is shown under Fig 4

SVM performs better than the LDA method on binary classification of the digits pair $1, 0$ and $4, 9$ and the kernel function is set to "Gaussian". 0.9% of the digit 4 is misclassified as 9 and only 3.9% of the digits 9 is misclassified as 4 on the test set, shown in Fig 7. For the separation between $0, 1$, non of the zero is classified as 1 and only 0.2% instances of 1 is mis-classified(Fig 6). The SVM performs better compare to the LDA methods in both regards.

The Decision tree method performs as good as the SVM models on the $4, 9$ separation but worse on the $0, 1$ separation, this is shown in Fig 9 and Fig 10. The DTree misclassified 0.4% and 0.7% of the "0" as "1" and "1" as "0" on the test set. For $4, 9$, it misclassified 13.9% of 4 as 9 and 14.6% of 9 as 4, higher than training set, indicating a potential **over fit** for the decision tree.

4.3 Performance Comparison on 10 Digits Classifications

Two models that has been benchmarked on all 10 digits are the SVM and the LDA which is shown in Fig 5 and Fig 8. Additional testing in the source code indicates that the QDA classifier performs as well as the SVM model. One explanation is that the distribution of samples of each class doesn't have the same amount of variance because LDA is not suited for classes with different amount of variance. Finally, the worst performance of LDA is due to under fitting because it has the same performance across all 3 of the test set and the training set.

5 Summary and Conclusions

The performance improvement on the accuracy and speed for using PCA as a dimensionality reduction technique is drastic. It's implement as extra in the source code, which we demonstrated that the decision tree become useless if the raw digits inputs are use to train the model. One can surmise the improvement if we made the choice of using LDA to find the best subspace on the data projected onto the PCA.

In addition, the importance to rescale the data into the $[0, 1]^n$ unit box in the positive quadrant for SVM is crucial in practice. This is required or the SVM to compute efficiently with a accuracy.

The decision tree model performs decently well only if it's trained on the projection onto the principal modes, which reflects the fact that Decision tree is sensitive to noise and pron to over-fit. To overcome this, one can consider using random forest to improve the accuracy, or Regression tree with a dimensionality reduction technique such as the PCA, or the LDA.

References

[1] Dr. Guanliang Checn. *Math 253: Linear Discriminant Analysis (LDA)*. 2020. URL: https://www.sjsu.edu/faculty/guangliang.chen/Math253S20/lec11lda.pdf.

[2] Jose Nathan Kutz. *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013, p. 387.

[3] *MathWorks: Help Cneter ¿ Documentation ¿ fitctree*. Accessed: 2010-09-30. URL: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.

[4] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[5] J.Nathan Kutz Steven L. Brunton. *Data-Driven Science and Enineering*. Cambridge University Press, 2019, p. 187.

[6] Joreme Friedman Trevor Hastie Robert Tibshirani. *The Elements of Statistical Learning: Data Ming, Inference, and Prediction 2nd Edition*. Springer, 2017, p. 419.

[7] Joreme Friedman Trevor Hastie Robert Tibshirani. *The Elements of Statistical Learning: Data Ming, Inference, and Prediction 2nd Edition*. Springer, 2017, p. 424.

5.1 Figures

Appendix A MATLAB Functions

1. ParameterPack:
   This is a class the contain all the relavent parameters for training the models.

2. MNISTPCA:
   This is a class that contain MNIST training and test data set, and it contains method for visualization of PCA, and the algorithms for PCA. It also perform filtering on the labels as well.

3. TreeMnist:
   This function trains a decision tree model on the MNIST data set.

4. TrainTestSplit:
   Splits the data sets and labels at random, 90% for training and 10% for validation.

5. SVMModelMnist:
   This methods trains a SVM model.

6. SplityByLabels:
   This method filter out digits given the data and the labels so they can be used to train binary models.

7. PrepraeData:
   This is a procedure that get calls before every model is trained, it transform the data according tha a class that stores the settings parameters.

8. mnist_parse:
   Given as a function that read the datainto matrices.

9. MinMaxStd:
   Transform that data so that is suitable for training SVM.

10. LDAModelMnist:
    Trains a LDA model using the setting parameters.

11. ImageNormalize:
    This function normalize the data such that they are zero mean and unit variance for all the images.

12. HWDemon:
    This is the script that produces all the computational results, with some extra code that is not discussed in this paper. Github link

A.1 ParameterPack

```matlab
classdef ParameterPack < handle
    % This class is going to contain all the things that is related to
    % graining, linear or quadratic LDA models.


    properties

        DisType;                        % Basic training settings
        PCAOnOff;
        SplitByLabels;
        PCAEnergyLevel;

        KernelFunc;                     % SVM specific settings.


        TrainX; TrainY;                 % Basic training data.
        TestX, TestY;
        ValX, ValY;


        TrainedModel;                   % Results after the training.
        CrossValLoss;
        PredictedValidateLabels;
        PredictedTestlabels;
        PredictedTrainingLabels;

    end

    methods
        function this = ParameterPack(splitByLabels, mode, pcaOnOff, pcaEnergy)
            % splitByLabels: The digits that we want the model to
            % distinguish.
            % SplitBylabels: The digits you want to filter out and train
            % the MNIST model for.
            % Mode:
            % The discriminant type you want to use to train the data.
            this.SplitByLabels = splitByLabels;
            this.DisType = mode;
            this.PCAOnOff = pcaOnOff;
            this.PCAEnergyLevel = pcaEnergy;
        end

        function this = visualizeResults(this, modelName)
            % Visualize the result of the trained model on all 3 data set,
            % and plot them, onto one figure.
            figure('Position', [0, 0, 2000 600]);
            subplot(1, 3, 1);
            confusionchart(this.PredictedTrainingLabels, this.TrainY, ...
                'RowSummary','row-normalized','ColumnSummary','column-normalized');
            title(strcat("Train Set ", modelName));

            subplot(1, 3, 2)
            confusionchart(this.PredictedValidateLabels, this.ValY, ...
                'RowSummary','row-normalized','ColumnSummary','column-normalized');
            title(strcat("Val Set ", modelName));

            subplot(1, 3, 3);
            confusionchart(this.PredictedTestlabels, this.TestY, ...
                    'RowSummary','row-normalized','ColumnSummary','column-normalized');
            title(strcat("Test Set ", modelName));

        end


    end
end
```

A.2 MNISTPCA

```
classdef MNISTPCA < handle
    % This class handle everything related PCA and PCA dimensionality
    % reduction.
    %   1. Store that original MNIST data matrix. Separated into test and
    %   validation sets.
    %   2. Store the SVD decomposition
    %   3. Produce beautiful plots.
    %   4. Peoject onto the subspace or return data traning data/test data
    %
    %   in the subspace projected onto.

    properties

        Data;
        DataStd;
        Labels;
        U; Sigma; V;

    end

    methods
        function this = MNISTPCA(n)
            % Make a PCA related stuff, on training set, or the test set.

            switch  n
                case 1
                    [images, labels] = mnist_parse...
                        ('data\train-images.idx3-ubyte', 'data\train-labels.idx1-ubyte');
                case 2
                    [images, labels] = mnist_parse...
                        ('data\t10k-images.idx3-ubyte', 'data\t10k-labels.idx1-ubyte');
                otherwise
                    error("Unknown Mode");
            end

            FlattendImagesStd = ImageNormalize(images);

            this.Data = images;
            this.DataStd = FlattendImagesStd;
            this.Labels = labels;

            [U, Sigma, V] = svd(FlattendImagesStd, "econ");

            this.U = U; this.Sigma = Sigma; this.V = V;

        end
```

```
function null = beautifulEnergyPlot(this)

    S = this.Sigma;

    figure;
    CulmulativeSigma = cumsum(diag(S))./sum(S, "all");
    SingularValues = diag(S)./max(diag(S));

    subplot(2, 1, 2);
    bar(CulmulativeSigma);
    Energy98 = find(CulmulativeSigma > 0.98, 1);
    Energy95 = find(CulmulativeSigma > 0.95, 1);
    Energy90 = find(CulmulativeSigma > 0.90, 1);
    xline(Energy98, '-', strcat("98%: r=", num2str(Energy98)),...
        "linewidth", 2, "fontsize", 12);
    xline(Energy95, '-', strcat("95%: r=", num2str(Energy95)),...
        "linewidth", 2, "fontsize", 12);
    xline(Energy90, '-', strcat("90%: r=", num2str(Energy90)),...
        "linewidth", 2, "fontsize", 12);
    title("Culmulative Energy");
    xlabel("Ranks");
    ylabel("Energy");

    subplot(2, 1, 1);
    bar(SingularValues(1:Energy98));
    title("Singular Values Spectrum");
    xlabel("Ranks");
    ylabel("Relative Size to max");

    null = 0;

end

function null = figurePlotProj3D(this)

    Sig = this.Sigma;
    labels = this.Labels;
    V = this.V;
    U = this.U;
    ModesIdx = [1, 5, 8];

    Projection = Sig*V.';
    Projection = Projection(ModesIdx, :);

    figure;
    for Label = 0:9
        disp(Label)
        PointsWithLabel = Projection(:, labels == Label);
        X = PointsWithLabel(1, :);
        Y = PointsWithLabel(2, :);
        Z = PointsWithLabel(3, :);
        Color = rand(1, 3);
        scatter3(X, Y, Z, 6,"MarkerFaceColor", Color, "MarkerEdgeColor", Color);
        hold on;
    end
    legend(["0", "1", "2", "3", "4", "5", "6", "7", "8","9"]);
    title("Principal Compoenent Projection Onto: 1, 5, 8");
    xlabel("Principal Mode: 1");
    ylabel("Principal Mode: 5");
```

```
            zlabel("Principal Mode: 8");

            null = 0;
        end

        function [DataProj, Labels, Projector, FilterIdx] = principalProj(this, principalEnergy, group)

            switch nargin
                case 1
                    principalEnergy = 0.9;
                    group = 0:9;
                case 2
                    group = 0:9;
                case 3

                otherwise
                    error("bleh")
            end

            EnergyS = cumsum(diag(this.Sigma))./sum(this.Sigma, "all");
            LastModeIdx = find(EnergyS > principalEnergy, 1);
            DataProj = this.Sigma*this.V.';
            DataProj = DataProj(1: LastModeIdx, :);
            [DataProj, Labels, FilterIdx] = SplitByLabels(DataProj, this.Labels, group);
            Projector = this.U.';
            Projector = Projector(1: LastModeIdx, :);

        end

    end
end
```

A.3 TreeMnist

```
function params = TreeMnist(mnistTraining, mnistTesting, params)



    % Train the model on the training set.
    % t = templateTree();
    % Dtree = fitrensemble(TrainX.', TrainY, 'Method', 'LSBOost', 'NumLearningCycles', 260, ...
    %      'Learners', t);

    [X, X1, TestX, Y, Y1, ~] = PrepareData(mnistTraining, mnistTesting, params);

    TrainedModel = fitctree(X.', Y);
    params.TrainedModel = TrainedModel;

    TrainPre = predict(TrainedModel, X.');
    params.PredictedTrainingLabels = TrainPre;

    ValPre = predict(TrainedModel, X1.');
    params.PredictedValidateLabels = ValPre;
    params.CrossValLoss = sum(ValPre ~= reshape(Y1, size(ValPre)))/length(ValPre);

    % Predict on the test set
    TestPre = predict(TrainedModel, TestX.');
    params.PredictedTestlabels = TestPre;
```

```
end
```

A.4 TrainTestSplit

```
function [X, X1, Y, Y1] = TrainTestSplit(data, label)
    % Permutes the data, 90% for training, and 10% for validation.
    Permvec = randperm(length(label));
    Sep = floor(0.9*length(label));
    PermvecTrain = Permvec(1: Sep);
    PermvecValidate = Permvec(Sep + 1: end);
    X = data(:, PermvecTrain);
    Y = label(PermvecTrain);
    X1 = data(:, PermvecValidate);
    Y1 = label(PermvecValidate);
end
```

A.5 SVMModelMnist

```
function params = SVMModelMnist(mnistTraining, mnistTesting, params)
    % This function split the trainig of the MNIST data into train and
    % validation set, and then it standardize the data using it to tran a
    % SVM model, and then it predict with the SVM model on all the data
    % set.
    %   The kernel function is going to be gaussian.
    % mnistTraing:
    %   An instance of the MNISTPCA class, has to be training instance
    % mnistTesting:
    %   An instance of the MNISTPCA class, has to be a test instance.
    % ldaParams:
    %   An instance of the ParameterPack class.

    [X, X1, TestX, Y, Y1, ~] = PrepareData(mnistTraining, mnistTesting, params);

    % Train the data on the training set:
    ModelTemplate = templateSVM("KernelFunction", params.KernelFunc);
    SVMModel = fitcecoc(X.', Y, "Learners", ModelTemplate);
    params.TrainedModel = SVMModel;

    % predict on the training set
    TrainPre = predict(SVMModel, X.');
    params.PredictedTrainingLabels = TrainPre;

    % predict on the val set
    ValPre = predict(SVMModel, X1.');
    params.PredictedValidateLabels = ValPre;
    params.CrossValLoss = sum(ValPre ~= reshape(Y1, size(ValPre)))/length(ValPre);

    % Predict on the test set
    TestPre = predict(SVMModel, TestX.');
    params.PredictedTestlabels = TestPre;

end
```

A.6 SplitByLabels

```
function [Data, Labels, FilteringIdx] = SplitByLabels(data, labels, splitby)
    % data is assume to be that each column is the sample, and each row corresponds to a certain feature.
    labels = labels.';
    BoolArray = zeros(size(labels));
    for L = splitby
       BoolArray = BoolArray + (labels == L);
    end
    BoolArray = boolean(BoolArray);
    Labels = labels(:, BoolArray);
    Data = data(:, BoolArray);
    FilteringIdx = find(BoolArray == 1);
end
```

A.7 PrepareData

```
function [X, X1, TestX, Y, Y1, TestY] = PrepareData(mnistTraining, mnistTesting, params)
    % Getting the testing data, training data for the MNIST data set.
    TrainX = mnistTraining.DataStd;
    TrainY = mnistTesting.Labels;
    TestX = mnistTesting.DataStd;
    TestY = mnistTesting.Labels;
    [TestX, TestY, ~] = SplitByLabels(TestX, TestY, params.SplitByLabels);
    [TrainX, TrainY, ~] = SplitByLabels(TrainX, TrainY, params.SplitByLabels);

    % If PCA, then project the data onto the PCA components;
    if params.PCAOnOff
        [TrainX, TrainY, Proj] = mnistTraining.principalProj(params.PCAEnergyLevel, params.SplitByLabels);
        TestX = Proj*TestX;
    end

    % minmax all features on test and training data.
    TrainX = MinMaxStd(TrainX);
    TestX = MinMaxStd(TestX);
    params.TestX = TestX;
    params.TestY = TestY;

    % Split the training into training and validation set.
    [X, X1, Y, Y1] = TrainTestSplit(TrainX, TrainY);
    params.TrainX = X;
    params.TrainY = Y;
    params.ValX = X1;
    params.ValY = Y1;
end
```

A.8 mnist_parse

MinMaxStd

```
function [DataStd] = MinMaxStd(data)
    data = data - min(data, [], "all");
    DataStd = data./max(data, [], "all");
end
```

A.9 LDAModelMnist

```
function params = LDAModelMnist(mnistTraining, mnistTesting, params)
    % What it does:
    %   Function us a part of the data to train the LDA model, and then use
    %   the rest to validate the model, and then it will predict with the
    %   model on the: Training set, the test set, and the  validation set.
    %
    % mnistTraining:
```

```
    %   An instance of the class LDAModelMnist, it has to be a training
    %   instace.
    % mnistTesting
    %   An instacoe of the class LDAModeMnist, it has to be a testing
    %   instance.
    % ldaParams:
    %   An instance f the class LDAParameters. Which is stored for all the
    %   parameters that are relatvent to the training of the model.

    [X, X1, TestX, Y, Y1, ~] = PrepareData(mnistTraining, mnistTesting, params);

    % Train the model on the 90% of the data and the labels:
    TrainedModel = fitcdiscr(X.', Y.', "DiscrimType", params.DisType);
    params.TrainedModel = TrainedModel;

    % Get the predicted labels on all the trainning set.
    params.PredictedTrainingLabels = predict(TrainedModel, X.');

    % Validate on the validation set, store the results for the
    % validation set.
    PredictedValidateLabels = predict(TrainedModel, X1.');
    params.PredictedValidateLabels = PredictedValidateLabels;
    params.CrossValLoss = sum(PredictedValidateLabels ~= reshape(Y1, size(PredictedValidateLabels)))...
        /length(PredictedValidateLabels);

    % Test on the test set, store the results for the test set.
    params.PredictedTestlabels = predict(TrainedModel, TestX.');



end
```

A.10 ImageNormalize

```
function FlattendImagesStd = ImageNormalize(images)
    FlattendImages = ...
        reshape(images, [size(images, 1)*size(images, 2), size(images, 3)]);
    FlattendImages = double(FlattendImages);
    FlattendImagesStd = FlattendImages - mean(FlattendImages, 1);
    FlattendImagesStd = FlattendImagesStd./std(FlattendImagesStd, 1);
end
```

A.11 HWDemon

```
%% Read it.
MnistTrain = MNISTPCA(1);
MnistTest = MNISTPCA(2);

%% Plot it.
MnistTrain.beautifulEnergyPlot();
saveas(gcf, "singular-value-spectrum", "png");
MnistTrain.figurePlotProj3D();
saveas(gcf, "3-mode-projection", "png");

%% LDA PCA ALL DIGITS
clc;
ldaParams = ParameterPack(0:9, "linear", true, 0.5);
LDAModelMnist(MnistTrain, MnistTest, ldaParams);
ldaParams.visualizeResults("LDA");
```

```
saveas(gcf, "conmat-lda-alldigits-with-pca", "png");

%% LDA PCA 4, 5, 7
clc;
ldaParams = ParameterPack([4, 5, 7], "linear", true, 0.5);
LDAModelMnist(MnistTrain, MnistTest, ldaParams);
ldaParams.visualizeResults("LDA");
saveas(gcf, "conmat-lda-digits-with-pca", "png");

%% LDA PCA, 4, 9
clc;
ldaParams = ParameterPack([4, 9], "linear", true, 0.5);
LDAModelMnist(MnistTrain, MnistTest, ldaParams);
ldaParams.visualizeResults("LDA");
saveas(gcf, "conmat-lda-2-hardests-digits-pca", "png");

%% LDA PCA, 0, 1
clc;
ldaParams = ParameterPack([0, 1], "linear", true, 0.5);
LDAModelMnist(MnistTrain, MnistTest, ldaParams);
ldaParams.visualizeResults("LDA");
saveas(gcf, "conmat-lda-2-easiest-digits-pca", "png");

%% SVM All Digits
clc;
SVMParams = ParameterPack(0:9, [], true, 0.5);
SVMParams.KernelFunc = "Gaussian";
SVMModelMnist(MnistTrain, MnistTest, SVMParams);
SVMParams.visualizeResults("SVM");
saveas(gcf, "conmat-svm-alldigits-pca", "png");

%% SVM PCA 4, 9 SPLIT
clc;
SVMParams = ParameterPack([4, 9], [], true, 0.5);
SVMParams.KernelFunc = "Gaussian";
SVMModelMnist(MnistTrain, MnistTest, SVMParams);
SVMParams.visualizeResults("SVM");
saveas(gcf, "conmat-svm-hardest-2-digits-pca", "png");

%% SVM PCA 0, 1 SPLIT
clc;
SVMParams = ParameterPack([0, 1], [], true, 0.5);
SVMParams.KernelFunc = "Gaussian";
SVMModelMnist(MnistTrain, MnistTest, SVMParams);
SVMParams.visualizeResults("SVM");
saveas(gcf, "conmat-svm-easiest-2-digits-pca", "png");

%% TREE PCA 4, 9
clc;
TreeParams = ParameterPack([4, 9], [], true, 0.5);
TreeMnist(MnistTrain, MnistTest, TreeParams);
TreeParams.visualizeResults("Dtree");
saveas(gcf, "conmat-dtree-hardest-2-digits", "png");

%% TREE PCA 0, 1
clc;
TreeParams = ParameterPack([0, 1], [], true, 0.5);
TreeMnist(MnistTrain, MnistTest, TreeParams);
TreeParams.visualizeResults("Dtree");
```

```
saveas(gcf, "conmat-dtree-easiest-2-digits", "png");

%% EXTRA MODELS THAT ARE NOT REQUIRED FOR THE HW %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TREE PCA ALL DIGITS
clc;
TreeParams = ParameterPack(0:9, [], true, 0.5);
TreeMnist(MnistTrain, MnistTest, TreeParams);
TreeParams.visualizeResults("Dtree");
saveas(gcf, "extra-tree-alldigits-pca", "png");

%% QDA PCA ALL DIGITS
clc;
ldaParams = ParameterPack(0:9, "quadratic", true, 0.5);
LDAModelMnist(MnistTrain, MnistTest, ldaParams)
ldaParams.visualizeResults("LDA");
saveas(gcf, "extra-qda-alldigits-pca", "png");

%% TREE ALL DIGITS
clc;
TreeParams = ParameterPack(0:9, [], false, 0.5);
TreeMnist(MnistTrain, MnistTest, TreeParams);
TreeParams.visualizeResults("Dtree");
saveas(gcf, "extra-dtree-nopca-alldigits", "png");

%% TREE ALL DIGITS LOW PCA 0.3
clc;
TreeParams = ParameterPack(0:9, [], true, 0.3);
TreeMnist(MnistTrain, MnistTest, TreeParams);
TreeParams.visualizeResults("Dtree");
saveas(gcf, "extra-dtree-lowpca-alldigits", "png");

%% SVM All DIGIT HIGH PCA 0.8
clc;
SVMParams = ParameterPack(0:9, [], true, 0.8);
SVMParams.KernelFunc = "Gaussian";
SVMModelMnist(MnistTrain, MnistTest, SVMParams);
SVMParams.visualizeResults("SVM");
saveas(gcf, "svm-alldigits-gaussian", "png"); v
```

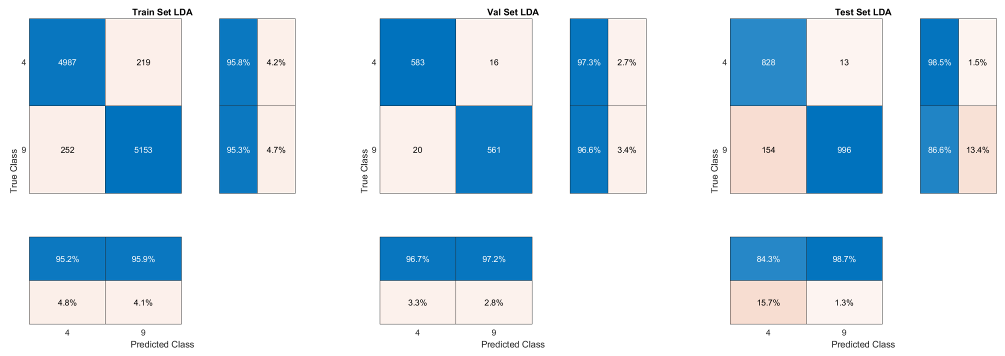

Figure 2: Figure 2: Hardest 2 Digits

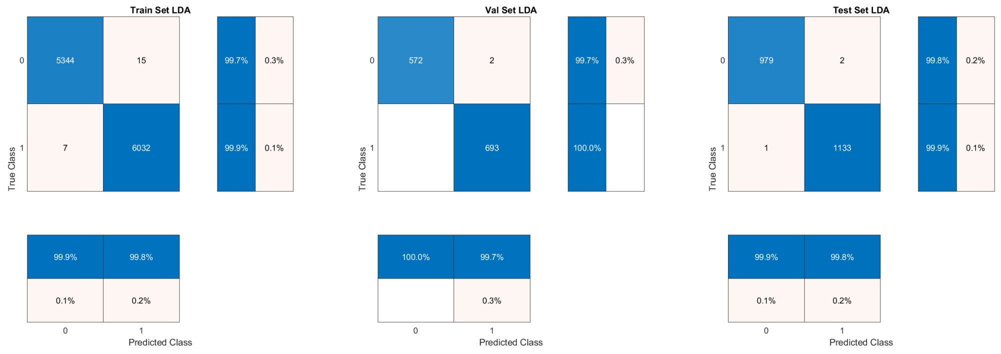

Figure 3: Figure 3: Easiest 2 Digits

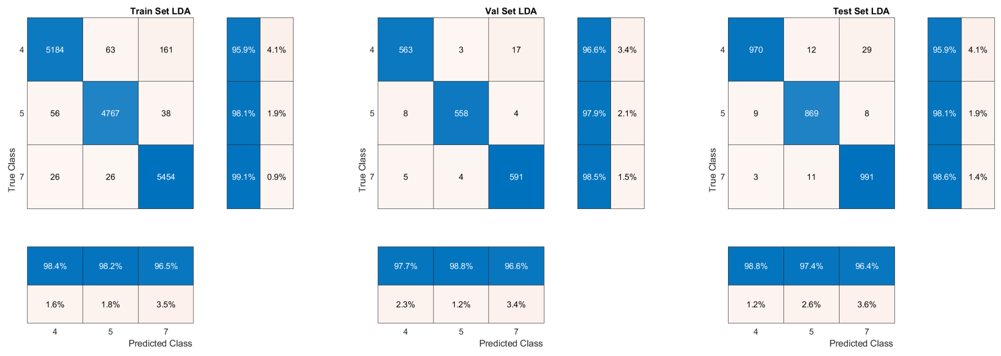

Figure 4: Figure 4: 3 Digits LDA

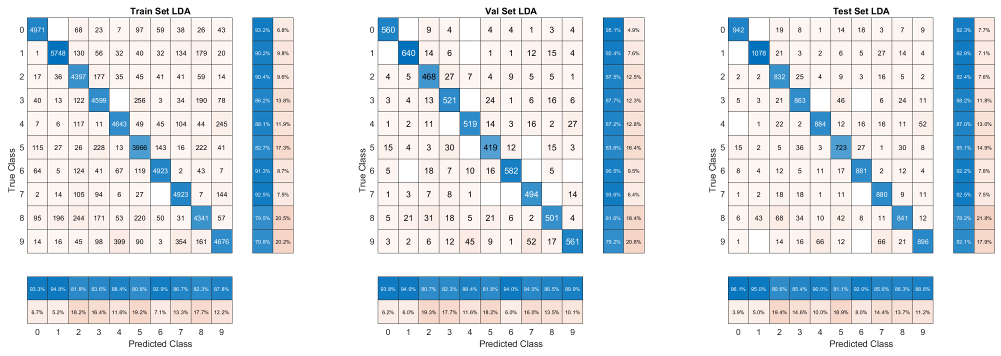

Figure 5: Figure 5: All Digits LDA

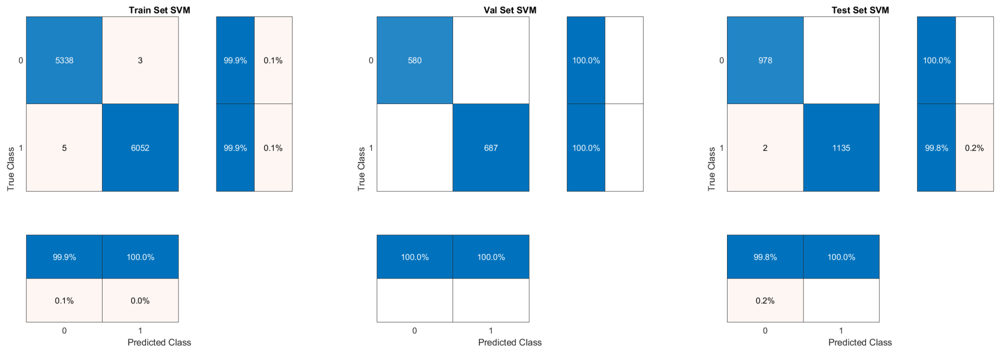

Figure 6: Figure 6: SVM on Easier 2 Digits

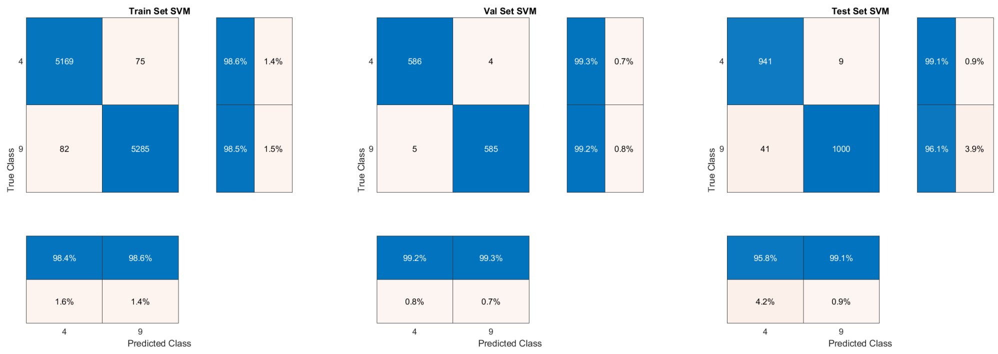

Figure 7: Figure 7: SVM on hardest 2 Digits

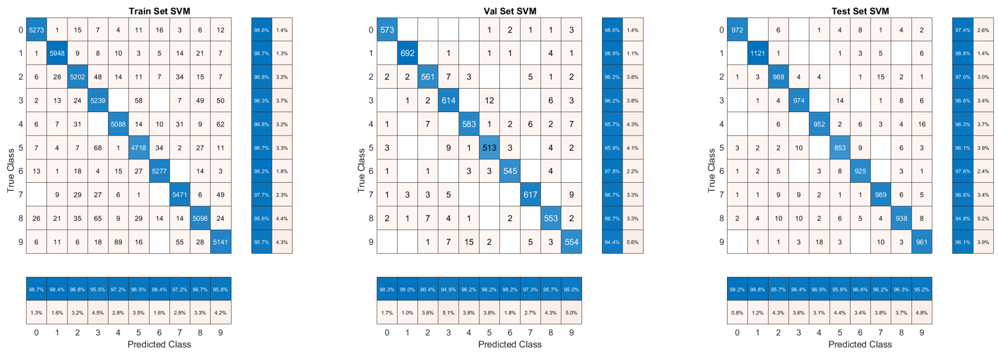

Figure 8: Figure 8: SVM on all 10 Digits

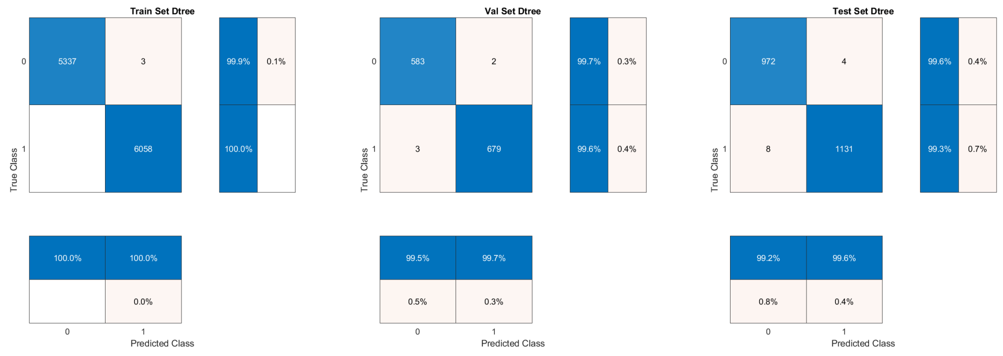

Figure 9: Figure 9: DTree on Easier 2 Digits

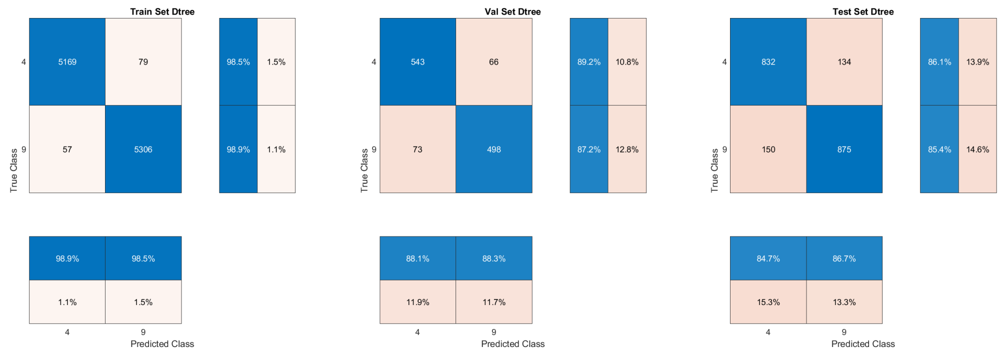

Figure 10: Figure 10: DTree on Hardest 2 Digits