Comparison Of LDA, SVM & Decision Tree Classifiers On MNIST Dataset

Arnold Peter Ruymgaart*
University of Washington
apruymgaart@gmail.com githubudemy

March 20, 2021

Abstract

In this report we analyze the MNIST handwritten digit dataset. We start with Principal Component Analysis (PCA) for dimensionality reduction and visualization. Visualization of the digits is accomplished by projection on 3 principal components such that each image is represented by a single 3D point. We next build a few supervised classifiers and compare their accuracy. Classification methods explored in this report include Linear Discriminant Analysis (LDA), Support Vector Machine (SVM) and Classification And Regression Tree (CART) used in the classification mode.

I. Introduction & Overview

In this study we compare the performance of a variety of supervised classification methods on the MNIST handwritten digit dataset. This popular data set consists of 60000 28×28 pixel training images of hand written digits along with an additional 10000 validation images. The MNIST data includes labels for both sets and is often used in performance benchmarks.

We'll briefly review the necessary theory to implement a few basic classification methods in Section II. Then in Section III we'll explain the code implementation in detail. The performance of our classification methods on the MNIST data set are included in Section IV.

II. Theoretical Background

The following principles are needed and briefly reviewed:

A. Clustering & classification
B. Rayleigh quotients & eigenvalue problems
C. PCA
D. LDA
E. SVM
F. Decision tree classifier

(A) Clustering & classification. The main task at hand is the construction of a classifier. A classifier assigns labels to objects. So classification is the task of correctly labeling an object by examining its features. This task requires some sort of object recognition.

*Contact for further information

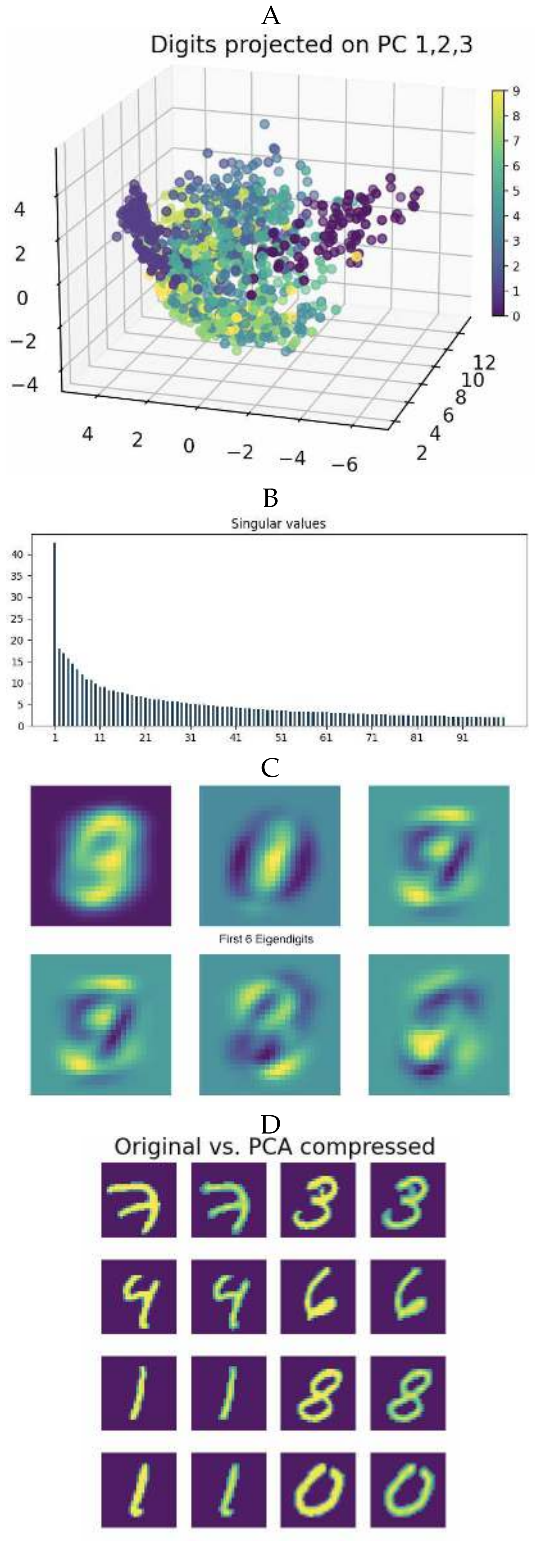
The PCA experiment we start with allows the visualization of the 10 digit classes by projection onto 3 principal components allowing a 3D scatter plot in which each digit is a single point. The digit points are color coded based on the provided labels in the MNIST data set. (Figure 1).

At this point we have only visualized the problem by reducing its (feature) dimensionality and picking three of those dimensions to plot the data. The labels used to color the digit points were given. The next task is to implement an algorithm that can assign a digit label of the test/validation set without prior knowledge of it but given the training set with its corresponding labels (supervised learning). The known labels of the test set are the used to see how accurate the label assignment was.

Binary classification implies there are only 2 possible options. When there are more than 2 label possibilities, the classification problem becomes multinomial AKA multi-class. In case of the MNIST data set, we have a multinomial classification problem because there are 10 possible digits. A set of binary classifiers can be combined to produce a multinomial classifier. In the MNIST data example, we essentially build 10 classifiers, one binary classifier for each digit vs. all others. We will look at pairwise classification in this report in order to quantify the difficulty of separating any two individual digits. But we note that pairwise classification is not the same as separating one digit from all others. We will look at the latter task as well.

(B) Rayleigh quotients & eigenvalue problems.

Figure 1: *PCA of MNIST images*



A Random sample of 1000 digits projected onto the first 3 principal components (V modes). Images are color coded by label. The label corresponds to the digit: 0 is a handwritten zero. **B** First 100 singular values. **C** The first 6 "eigendigits". The projections in panel A are the weights of linear combinations of the top row of panel C. **D** Images reproduced (on the right, true left) from compression (onto first) 100 PC.

The Rayleigh quotient is defined as the following scalar valued function of a vector argument:

$$r(\vec{x}) = \frac{\vec{x}^* A \vec{x}}{\vec{x}^* \vec{x}} \tag{1}$$

where if $\vec{x}$ happens to be an eigenvector of A, the above reduces to $r = \lambda \frac{\vec{x}^* \vec{x}}{\vec{x}^* \vec{x}} = \lambda$ so the Rayleigh quotient of an eigenvector is the corresponding eigenvalue. The **Generalized Rayleigh quotient** has a matrix in the denominator as well. If the matrices A and B are symmetric, these vector-matrix-vector products, e.g. $\vec{x}^* A \vec{x}$ are called quadratic forms. The generalized Rayleigh quotient:

$$r(\vec{x}) = \frac{\vec{x}^* A \vec{x}}{\vec{x}^* B \vec{x}} \tag{2}$$

The latter equation is related to the **generalized eigenvalue problem**:

$$A\vec{x} = \lambda B \vec{x} \tag{3}$$

(C) PCA. We can write PCA (SVD of data matrix and eigendecomposition of covariance) in terms of Rayleigh quotient. Singular vectors are solutions of:

$$argmax_{\vec{x}} = \frac{\vec{x}^* Cov \vec{x}}{\vec{x}^* \vec{x}} \tag{4}$$

We find unit $v = \frac{x}{||x||}$ that maximizes $||A\vec{v}||_2$ and therefore also $||Av||_2^2$. Proof: $||A\vec{v}||_2^2 = (A\vec{v})^T A\vec{v} = \vec{v}^T(A^T A)v = v^T Cov \vec{v} = \vec{x}^* Cov \vec{x} / ||\vec{x}||^2$ and the latter equals Eqn. 4 with $||\vec{x}||^2 = \vec{x}^* \vec{x}$. So the solution singular vectors point along the direction that maximizes the variance and therefore is an optimal basis vector in which to represent the data.

In PCA, one image is seen as one random vector of (in this case) $28^2 = 784$ random variables. The 60000 are samples of this random vector. The variance-covariance matrix (of a random vector) is the outer product:

$$E[(\vec{X} - \vec{\mu}_X)(\vec{X} - \vec{\mu}_X)^T] \tag{5}$$

For detailed additional information, see Appendix C.

The eigenvectors of Cov can be found (we can solve Eqn. 4) by SVD of the centered data matrix $A_c = U\Sigma V^*$. The columns of V (right-singular vectors, rows of V^*) are eigenvectors of $A_c^* A_c$. The columns of U (left-singular vectors) are eigenvectors of AA^*. Proof: with $A = U\Sigma V^*$ we have $A^*A = (U\Sigma V^*)^*(U\Sigma V^*) = V\Sigma U^* U\Sigma V^* = V\Sigma^2 V^*$

(D) Linear Discriminant Analysis (LDA). LDA is a projection method just like PCA. But the optimization problem is different as will be shown below. In PCA we found an optimal basis in which to represent the data. Here, we find a basis in which class separation is maximal. The LDA objective function is a generalized Rayleigh quotient:

$$\vec{w} = argmax_{\vec{w}} \frac{\vec{w}^T S_B \vec{w}}{\vec{w}^T S_w \vec{w}} \tag{6}$$

In LDA, the symmetric matrices S_B and S_W are the between class and within-class scatter matrices. Eigenvector $\vec{w}$(of the generalized eigenvalue problem) is the solution. A **scatter matrix is related to a covariance matrix**. In this case we will look a multiple random vectors, one for each class. Like the covariance, the scatter matrices can be written as outer products:

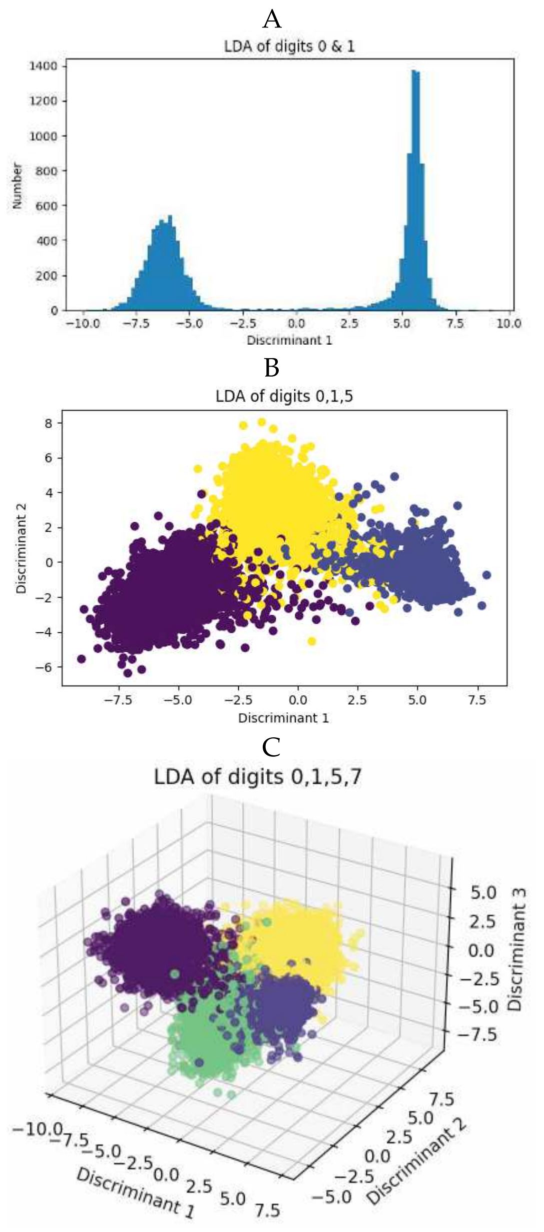
$$S_B = \sum_{c=1}^{N_{classes}} (\vec{\mu}_c - \vec{\mu})(\vec{\mu}_c - \vec{\mu})^T \tag{7}$$

$$S_W = \sum_{c=1}^{N_{classes}} \sum_{\vec{x}} (\vec{x} - \vec{\mu}_c)(\vec{x} - \vec{\mu}_c)^T \tag{8}$$

A **linear discriminant** is a solution to the optimization problem Equation 6 and is a linear combination of input (features). Each discriminant is a vector that can separate two classes. In a multiclass, this is a separation of one class vs. all other (viewed as the other class). If we project the data onto this discriminant each data element will be a point along this line. Each class will form a cluster on this line. The LDA optimization problem defined in Equation 6 leads to eigenvalue equation $S_B\vec{w} = \lambda S_W \vec{w}$ and ensures that the means of the clusters are separated maximally. In panel A of Figure 2 we see a histogram of the data points of digits 0 and 1 projected onto the resultant single discriminant vector. There is a *decision point* along each discriminant between the means of the two classes (distances between interclass data points). In a multiclass separation, N classes will result in $N-1$ discriminant vectors.

(E) Suport Vector Machine (SVM). We can think of an SVM binary classifier as a **hyperplane** that divides the two data classes.

Figure 2: *LDA of MNIST images*



A Histogram of projection of training set digits 0 & 1 onto the single resulting discriminant. **B** Projection of training set digits 0,1 & 5 onto the resulting 2 discriminants. **C** Projection of training set digits 0,1,5 & 7 onto the 3 discriminants. The 4 classes resulted in $N-1=3$ linear discriminant functions onto which the training set digits are projected. The discriminants are linear combinations of features.

Consider the equation for a (hyper)plane with normal vector $\vec{n}$. The hyperplane is the set $\{\vec{x} \mid h(\vec{x}) = 0\}$ where

$$h(\vec{x}) = \vec{n}^T \vec{x} - b \tag{9}$$

In a linearly separable 2 class example, there will be some distance perpendicular to the dividing surface that separates the nearest point from each class. We'll say data of the first class has label 1 and of the second class -1. We can in fact draw *two* parallel hypersurfaces: one that touches the nearest point(s) from one class on its surface and the other with the nearest point(s) from the other class on its surface. The **support vectors** are input vectors that touch the edge of the boundary region between the two hyperplanes. That means they are on one of the hypersurfaces. *These support vectors are sufficient to solve the problem.* The two hyperplanes are defined as follows: hyperplane 1 : if label $y_i = 1$ then $\vec{w}^T \vec{x}_i - b \geq 1$. Hyperplane 2 : if $y_i = -1$ then $\vec{w}^T \vec{x}_i - b \leq -1$. If either of the above inequalities is an equality for some $\vec{x}_i$ then that vector is a support vector. Together, for all i we have $y_i(\vec{w}^T \vec{x} - b) \geq 1$

The distance between these two parallel surfaces is along the normal vector and equals $2/||\vec{w}||$. Therefore, minimizing $||\vec{w}||$ will maximize the distance between the hyperplanes. Maximal distance between the hyperplanes ensures the surfaces are angled optimally for separation of the classes. So solving SVM is solving the (quadratic programming) optimization problem:

$$\min ||\vec{w}|| \; s.t. \; y_i(\vec{w}^T \vec{x} - b) \geq 1, \quad i = 1, \dots, n \tag{10}$$

Classification is now done by checking which side of the plane some new vector is on.

Nonlinear SVM: SVM is inherently linear but can be made non-linear by generalizing the dot product with the "kernel trick". Distances are calculated with the dot product. This allows use of the kernel trick with kernel k [generalization of the dot product] $\vec{x}^T \vec{y} \rightarrow k(\vec{x}^T \vec{y}) = \langle \phi(\vec{x}), \phi(\vec{y}) \rangle$. The hyperplanes in the higher-dimensional space are defined as the set of points whose dot product with a vector in that space is constant.

Multiclass SVM. This can again be done by making multiple one vs all others classifiers. In case of MNIST, there would be one binary classifier for each digit. E.g. the classifier for the 0 digit is binary: is the digit 0, yes/no.

Rayleigh quotients in SVM. With the preceding discussion about Rayleigh quotients in mind one may wonder, can the SVM optimization problem also be defined in terms of a (generalized) Rayleigh quotient? Indeed, a multi-surface variant can be [Mangasarian, 2005].

(F) Decision tree classifier. A decision tree is usually a binary tree. The simplest binary tree is a single root node with two leaves (child nodes). Each leave can have its own child nodes to form a more complex structure. The number of generations in such a tree is called the number of levels or depth.

In a classifier, each root (or node) in the tree makes a binary "decision" based on one input feature. With all numerical input such as the case here, the decision is a cutoff value of the feature. For example, if the feature is age and the cutoff is 40 then all data samples with the age < 40 will end up in one child node while those with age ≥ 40 will end up in the other. This example is the simplest possible classifier, in this case the classification may be young or old based on just one feature.

In a two feature example, we may now imagine a classifier that predicts the medical condition of diabetes

Table 1: *LDA % Error In Pair-wise Digit Separation*

	1	2	3	4	5	6	7	8	9
0	0.3	1.3	0.5	0.3	1.1	1.1	0.8	1.0	0.9
1		1.6	0.7	0.3	0.6	0.3	1.0	2.0	0.6
2			2.4	1.9	2.2	2.4	2.1	3.2	1.7
3				0.7	3.4	0.5	1.8	3.7	2.0
4					1.2	1.0	1.5	1.0	4.9
5						2.5	0.9	4.7	1.6
6							0.2	1.3	0.4
7								1.6	4.0
8									2.4
9									

Table 2: *SVM % Error In Pair-wise Digit Separation*

	1	2	3	4	5	6	7	8	9
0	0.1	0.8	0.5	0.4	1.0	1.0	0.4	0.8	0.9
1		0.8	0.4	0.1	0.6	0.2	0.6	0.7	0.5
2			2.0	1.8	1.6	1.7	1.5	2.0	1.2
3				0.6	3.5	0.6	1.4	3.2	1.6
4					0.8	0.8	0.7	0.7	3.3
5						1.9	1.0	4.5	1.1
6							0.2	0.9	0.5
7								0.9	3.6
8									1.9
9									

Table 3: *CTree % Error In Pair-wise Digit Separation*

	1	2	3	4	5	6	7	8	9
0	0.4	2.7	1.6	1.4	3.3	3.8	1.2	2.7	2.5
1		1.2	1.3	0.8	1.0	1.1	1.5	1.7	0.9
2			5.1	3.4	4.2	3.9	3.7	5.8	3.3
3				2.0	5.4	2.1	3.4	7.7	3.9
4					2.5	3.1	3.7	3.8	9.1
5						4.0	2.7	7.9	5.1
6							1.4	3.0	1.5
7								3.4	8.1
8									5.2
9									

Table 4: *Summary % Error In Pair-wise Digit Separation On Training Data*

	LDA	SVM	CTree
Best	(6,7) 0.197	(0,1) 0.0	(all) 0.0
worst	(3,5) 4.493	(3,5) 3.627	None 0.0

based on just two features: age and weight. The first node makes the young/old decision of the first example. But each child node "young" and "old" now has two child nodes of its own: "diabetes" or "no diabetes" based on the second feature which is weight. In the young node, the weight value that decides diabetes is higher than in the old node: the correlation between overweight and diabetes is more pronounced in older people. There is a single root node and there are 4 leaves with 2 possible leave values in this example. Each of the leaves is a diabetes outcome.

In the MNIST example, the leaves correspond to the digit classes. So there are 10 possible leave values. The decision tree classifier is a supervised nonlinear classifier.

III. Algorithm Implementation & Development

Although we could calculate the within- and between-class scatter matrices and solve the generalized eigenvalue problem as described in Section II D to build an LDA classifier, we choose to utilize the *lda* method provided by Python-sklearn. Similarly, we'll use Python-sklearn's SVC (Support Vector Classifier) and $DecisionTreeClassifier$. Therefore the focus of this implementation is on processing the MNIST dataset (compression with PCA) and do comparative analysis.

The algorithm implementation consists of a number of separate scripts. Several scripts implement functions and these are included in Appendix A.

The function $allPairs$ returns all unique $2-$combinations where $i \neq j$. In the case of 10 digits, there are 45 pairs.

The function *slectDigits* returns a subset of of images and labels that excludes all that are not in the input *digits* list. The output is ordered by digit label. The *size* variable is the length of a flattened data vector. For example, a single MNIST image $size = 28^2$.

```
**slectDigits(digits, M, labels)**
    $R, L, size = [], [], len(M[0].flatten())$
    for $j = 0 : len(digits)$ do:
        $\vec{i} = argwhere(labels == digits[j])$
        $R = R + list(M[\vec{i}].reshape(len(\vec{i}), size))$
        $L = L + \vec{1} \cdot digits[j]$
    end for
    return $R, L$
```

The **pca** function implements the PCA steps as described in Section II C utilizing the Python-numpy *svd*:

```
**PCA(A)**
    $A_c = np.zeros(A.shape)$
    for $k = 0 : A.shape[0]$ do
        $A_c[k,:] = A[k,:] - average(A[k,:])$
    end for
    $A_c = \frac{A_c}{\sqrt{A.shape[1]-1}}$
    $[U, S, V] = svd(A_c)$
    $U_r = U[:, 0 : nv]$
    $P = A_c^T U_r$
    return $U_r, S, A_c, P$
```

Then there are two main scripts that carry out the tasks of reading the MNIST data carrying out the various analysis. These main scripts are included in Appendix B.

The script $hw4.py$ is the main analysis script. I reads in the MINIST dataset and optionally reduces its dimensionality through PCA. The *slectDigits* function is then called a number of times to make training and testing sets of 2, 3, 4 or all 10 digits. It then trains LDA, SVM and CTree classifiers. The classification errors are determined by generating **confusion matrices**.

The $allpairs.py$ script reads the MNIST training and validation sets and stores those into the variables M_{train}, L_{train} and M_{test}, L_{test}. The script then optionally carries out PCA dimensionality reduction and subsequently calls the pairwise performance function. The *pairwise performance* function loops over the three methods (SVM, LDA and Tree) and all unique pairs obtained from the *allPairs* function. For each pair, the digits are selected by *slectDigits* and classification error is calculated for each method.

```
**Pairwise performance()**
    $pairs = allPairs(10)$
    for $method = 1 : 4$ do
        for $k = 0 : len(pairs)$ do
            $M_T, L_T = selectDigits(pairs[k], M_{train}, L_{train})$
            $M_V, L_V = selectDigits(pairs[k], M_{test}, L_{test})$
            if $method == 1$ then
                $model = lda(M_T, L_T)$
            else if $method == 2$ then
                $model = svm(M_T, L_T)$
            else if $method == 3$ then
                $model = dtree(M_T, L_T)$
            end if
            $predict = model.predict(M_V)$
        end for
        $error.append(predict - L_V)$
    end for
    store error in file $pairwiseError.dict$
```

IV. Computational Results

For all classification tasks, the set of MNIST images comes pre-divided into a training set and a validation set (as provided). The validation set was therefore new to the classifiers: the validation images had not been "seen" before during training.

Results of **PCA** of the MNIST dataset are summarized if Figure 1. A random sample of 1000 digits was projected onto the first 3 PC in panel A.

Rank. The first singular value is clearly dominant (see panel B Figure 1). But the tail continues beyond 100, rank is difficult to determine. If we keep 100 components, the images reproduced are of such quality that it is difficult to say which is the original (see panel D of Figure 1). With 10 features, the prediction errors of all three methods are significantly higher (highest near 20%, results not included). So 10 PC features is insufficient. With too few features, there is also less difference between methods: the advantage of SVM/Ctree over LDA is less pronounced.

Interpretation of the SVD matrices. $A = U\Sigma V^* \implies U^T A = \Sigma V^*$ and $AV = U\Sigma$. Since unitary, U and V can be viewed as rotations while Σ performs a scaling. The columns of V (right-singular vectors, rows of V^*) are eigenvectors of A^*A. The columns of U (left-singular vectors) are eigenvectors of AA^*. Here, $A = 784 \times 60000$ is the MNIST image set with linearized images as column vectors of length $28^2 = 784$. So columns of U *are eigenvectors of pixel covariance* $\equiv$ *eigendigits = optimal basis*. If an image is random vector $\overrightarrow{X}$ (a vector of 784 pixel random vars e.g. X_1 is $pixel[0,0]$) then each column is a sample of $\overrightarrow{X}$ see Eqn. 14 in Appendix C. ΣV^T is the (transpose) projection of data A onto optimal eigendigit basis.

LDA separation of 2,3 or 4 digits. The number of linear discriminants is $N - 1$ where N is the number of classes. This allows visualization of LDA by projection on discriminants for up to 4 classes and this is visualized in Figure 2.

Pairwise digit separation (classification) using LDA, SVM & CTree. The accuracy of the three methods was assessed in its ability to correctly classify all pairs of digits from random samples of the test set. The classifier had not "seen" the test set before. The pairwise accuracy was determined after PCA dimensionality reduction on 100 principal components. There are a total of 45 unique pairs of the set of digits $\{0,1,2,3,4,5,6,7,8,9\}$. We can visualize the pairs in a 10×10 symmetric matrix, taking the upper (or lower) triangle and removing the diagonal. These equal the error matrices shown for all pairs in Tables 1, 2 and 3.

Performance on the training set. Similar results were obtained for the training data sets (how well did the classifier perform on classification of the same set it was trained on). A summary of classification error on the training set is listed in Table 4. For any machine learning method, it should be easier to predict the set the method is trained on than a new set the method hasn't seen before. In case of the classification tree (CTree), all training set prediction error is zero: the CTree perfectly learns the training set *in no max depth is set*. This default causes classification to be prone to *overfitting*.

The most difficult digit pairs to separate are digits that look alike: $(3,5)$, $(3,8)$, $(4,9)$, $(5,8)$ and $(7,9)$ highlighted red in Tables 1, 2 and 3. **Easiest digit pairs to separate** (highlighted green) are $(0,1)$, $(0,4)$, $(1,4)$, $(1,6)$ and $(6,7)$. **Separation of all 10 digits**: the confusion matrices for 10 digit separation by LDA and SVM are included in Appendix D. When separating one vs. all other classes, the limitations come from the most difficult to separate pairs. Results in the confusion matrices are similar to the pairwise results given.

V. Summary and Conclusions

We compared 3 supervised classifiers: LDA,SVM and CTree (CART). LDA is the only strictly linear method and is expected to perform less well than SVM or CART. The SVM in this report was run in linear mode (no kernel trick was used). We can see from projection onto linear discriminants (Figure 2) that linear methods may not be sufficient to separate certain digits.

In this study, SVM outperformed both LDA and decision tree classifier, even in linear mode. LDA unexpectedly outperformed the CTree classifier. The tree method showed evidence of overfitting (zero training set error along with relatively high test set error). Testing max-depth limits resulted in some training set error but did not improve overall result. It may be possible to get better CTree results using different parameter settings. Experimenting with nonlinear kernels may also further improve SVM results. A total error rate of just 1.4% on MNIST test set classification is reported by [LeCun, web] using a Gaussian kernel SVM.

References

[Mangasarian, 2005] Mangasarian, O. & Wild E (2005). Multisurface Proximal Support Vector Machine Classification via Generalized Eigenvalues. *IEEE Transactions On Pattern Analysis And Machine Intelligence*, Vol. 27, No. 12, Dec. 2005

[LeCun, web] http://yann.lecun.com/exdb/mnist

A. Python functions used

PCA functions (pcaF.py) as described in Section III

```
import numpy as np
import sklearn
import sklearn.utils
import sklearn.utils.extmath

def pca(A, t=True, nv=3):
        As = np.zeros(A.shape)
        for k in range(A.shape[0]): As[k,:] = A[k,:] - np.average(A[k,:])
        As /= np.sqrt(float(A.shape[1]-1))
        M = As.T if t else As
        [U,S,Vt] = sklearn.utils.extmath.randomized_svd(M, nv)
        Ured = U[:,0:nv]
        P = np.matmul( np.diag(S), Vt ).T #P = np.matmul(A.T, Ured)
        print('input A shape',M.shape, 'U', U.shape, 'S', S.shape, 'Vt', Vt.shape, 'P', P.shape, 'Ured', Ured.shape)
        return Ured,S,P
```

Classifier functions (classifier.py) as described in Section III

```
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.tree import DecisionTreeClassifier
from sklearn import svm
from sklearn.svm import SVC

def selectDigits(digits, M, labels, sz=None):
        R, l = [],[]
        if sz is None: sz = len(M[0].flatten())
        for d in digits:
                ind = np.argwhere(np.array(labels) == d)
                R += (M[ind].reshape(len(ind),sz)).tolist()
                l += [d] * len(ind)
        return np.array(R), np.array(l)

def randperm(ListArr, nr=None):
        ret = []
        for j in range(len(ListArr)): ret.append([])
        indxs = np.random.permutation(len(ListArr[0]))
        if not nr is None: indxs = indxs[0:nr]
        for n,i in enumerate(indxs):
                for j in range(len(ListArr)):
                        ret[j].append( ListArr[j][i] )
        for j in range(len(ListArr)): ret[j] = np.array(ret[j])
        return ret

def lda(X, labels):
        print('LDA: inputs shape', X.shape, labels.shape)
        clf = LinearDiscriminantAnalysis()
        clf.fit(X, labels)
        print('LDA variance ratio', clf.explained_variance_ratio_)
        return clf

def svm(X, labels):
        clf = SVC(kernel='linear') # Linear Kernel, support vector classifier
        clf.fit(X, labels)
        return clf

def dtree(X, labels, mxDepth=None, mxFeat=None):
        clf = DecisionTreeClassifier(max_depth=mxDepth, max_features=mxFeat) #
        clf.fit(X, labels)
        return clf

def allPairs(n):
        pairs = []
        for i in range(0,n):
                for j in range(i+1,n):
                        pairs.append([i,j])
        return pairs

def confusion(predict, true):
        u1, u2 = list(np.unique(predict)), list(np.unique(true))
        ulab = np.sort( np.unique( np.array(u1+u2) ) )
        F = np.zeros((len(ulab),len(ulab)))
        labIndx = {}
        for n, lb in enumerate(ulab): labIndx[lb] = n
        print(labIndx)
        for k in range(len(predict)):
                i,j = labIndx[predict[k]], labIndx[true[k]]
                F[i,j] += 1
        return F, labIndx

def adjustImage(im, thresh=0.3):
        im = im - np.min(im)
        mx = np.max(im)
        im[im < thresh*mx] = 0
        return im/np.max(im)

if __name__ == '__main__':
        print( randperm( [[1,2,3,4,5], [4,5,6,7,8]] ) )
```

Plotting functions (plottingFunctions.py)

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

def adjustFigAspect(fig,aspect=1):
        xsize,ysize = fig.get_size_inches()
        minsize = min(xsize,ysize)
        xlim = .4*minsize/xsize
        ylim = .4*minsize/ysize
        if aspect < 1: xlim *= aspect
        else:          ylim /= aspect
        fig.subplots_adjust(left=.5-xlim, right=.5+xlim, bottom=.5-ylim, top=.5+ylim)

def rmWhitespace(axisOff=False):
        if axisOff:
                plt.gca().set_axis_off()
                plt.subplots_adjust(top = 1, bottom = 0, right = 1, left = 0, hspace = 0, wspace = 0)
        else:
                plt.subplots_adjust(hspace = 0, wspace = 0)

        if axisOff:
                plt.gca().xaxis.set_major_locator(plt.NullLocator())
                plt.gca().yaxis.set_major_locator(plt.NullLocator())
        plt.margins(0,0)

def plot3(X,Y,Z, c=None, ttl='', fname='', cbar=False, axLabels=None):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        #if ttl != '': plt.title(ttl)
        im = None
        if not c is None: im = ax.scatter(X, Y, Z, c=c)
        else: im = ax.scatter(X, Y, Z)
        if not axLabels is None:
                ax.set_xlabel(axLabels[0]), ax.set_ylabel(axLabels[1]), ax.set_zlabel(axLabels[2])
        if not ttl == '': ax.set_title(ttl)
        if cbar:
                fig.colorbar(im, shrink=0.7)
                #divider = make_axes_locatable(ax)
                #cax = divider.append_axes("top", size="5%", pad=0.05)
                #plt.colorbar(im, cax=cax)
        if fname == '': plt.show()
        else:
                plt.savefig(fname, dpi=200, bbox='tight')
                plt.clf()

def plotSV(S, fname=''):
        fig, ax = plt.subplots()
        plt.title('Singular values')
        if False: ax.bar(np.arange(len(S)), np.log(S))
        else: ax.bar(np.arange(len(S)), S, color=(0.1, 0.2, 0.3, 1.0), width=0.5)
        plt.savefig('images/exp2_sing_values.png', figsize=(8, 6), dpi=300, bbox_inches='tight')
        xlabs, xticks = [], []
        if len(S) < 50:
                xticks = np.arange(len(S))
                for j in range(len(S)): xlabs.append('PC%d' % (j+1))
        else:
                for j in range(len(S)):
                        if j % 10 == 0.0:
                                xlabs.append('%d' % (j+1))
                                xticks.append(j)
        ax.set_xticks(xticks)
        ax.set_xticklabels(xlabs)
        if fname == '': plt.show()
        else:
                plt.savefig(fname, figsize=(8, 6), dpi=300, bbox_inches='tight')
                plt.clf()

def plotImageMatrix(IMG, nrow, ncol, fname='', ttl=''):
        rmWhitespace(axisOff=True)
        fig, ax = plt.subplots(nrow, ncol)
        if ttl != '' : fig.suptitle(ttl)
        for j in range(nrow):
                for i in range(ncol):
                        indx = j*nrow + i
                        ax[j,i].axis('off')
                        ax[j,i].imshow(IMG[indx].reshape(28,28))
        if fname == '': plt.show()
        else:
                plt.savefig(fname, figsize=(8, 6), dpi=300, bbox_inches='tight')
                plt.clf()

def plotLDA(lda, labels, ttl='', fname=''):
        if lda.shape[1] == 1:
                if ttl != '': plt.title(ttl)
                plt.ylabel('Number'), plt.xlabel('Discriminant 1')
                plt.hist(lda.reshape(-1), bins=100)
        elif lda.shape[1] == 2:
                plt.xlabel('Discriminant 1'), plt.ylabel('Discriminant 2')
                if ttl != '': plt.title(ttl)
                plt.scatter(lda[:,0], lda[:,1], c=labels)#, cmap='rainbow', alpha=0.7, edgecolors='b')
        elif lda.shape[1] == 3:
                plot3(lda[:,0], lda[:,1], lda[:,2], c=labels, axLabels=['Discriminant 1','Discriminant 2','Discriminant 3'], ttl=ttl)
        else:
                print('Dimensions too high')
                return
        if fname == '': plt.show()
        else:
                plt.savefig(fname, figsize=(8, 6), dpi=300, bbox_inches='tight')
                plt.clf()
```

B. Python codes

Main script *hw4.py* as described in Section III:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorFiles import *
from pcaF import *
from plottingFunctions import *
from classifier import *

#==== MNIST data ====
M1,Mtest = tnsrFile2numpy('data/mnist/mnist_train_images.npz'), tnsrFile2numpy('data/mnist/mnist_test_images.npz')
M3,L1 = tnsrFile2numpy('data/mnist/mnist_validate_images.npz'), tnsrFile2numpy('data/mnist/mnist_train_labels.npz')
Ltest,L3 = tnsrFile2numpy('data/mnist/mnist_test_labels.npz'),    tnsrFile2numpy('data/mnist/mnist_validate_labels.npz')
Mtrain, Ltrain = np.array(list(M1) + list(M3)), np.array(list(L1)+list(L3)) # 60K training images
ltrain,ltest = [],[]
for vbin in Ltrain: ltrain.append(np.argmax(vbin))
for vbin in Ltest: ltest.append(np.argmax(vbin))
ltest,ltrain = np.array(ltest), np.array(ltrain)
print('Training set', Mtrain.shape, Ltrain.shape, len(ltrain), 'Validation set', Mtest.shape, Ltest.shape, len(ltest))

#==== PCA ====
nEv = 100
useReducedDims = True
Ured,S,P = pca(Mtrain, t=True, nv=nEv)
if useReducedDims:
        Mtrain = np.matmul(Mtrain, Ured)
        Mtest = np.matmul(Mtest, Ured)
        print('TRAIN matrix after PC projection (dim reduce)', Mtrain.shape)

#==== some training and validation sets ====
train_M01, train_l01     = selectDigits([0,1],   Mtrain, ltrain)
train_M12, train_l12     = selectDigits([1,2],   Mtrain, ltrain)
train_M89, train_l89     = selectDigits([8,9],   Mtrain, ltrain)
train_M015,train_l015    = selectDigits([0,1,5], Mtrain, ltrain)
train_M0157,train_l0157  = selectDigits([0,1,5,7], Mtrain, ltrain)
val_M12,    val_l12      = selectDigits([1,2],   Mtest, ltest)
val_M015,   val_l015     = selectDigits([0,1,5], Mtest, ltest)
[train_M12, train_l12]   = randperm([train_M12,  train_l12]) # randomize input order
[train_M89, train_l89]   = randperm([train_M89,  train_l89])
[train_M015,train_l015]  = randperm([train_M015, train_l015])
[train_M0157,train_l0157]= randperm([train_M0157, train_l0157])
[val_M12,   val_l12]     = randperm([val_M12,    val_l12])
[val_M015,  val_l015]    = randperm([val_M015,   val_l015])

#==== LDA ====
LD01 =  lda(train_M01,  train_l01)             # train LDA 2 digits, easy to separate
LD12 =  lda(train_M12,  train_l12)             # train LDA 2 digits
LD89 =  lda(train_M89,  train_l89)             # train LDA 2 digits, hard to separate
LD015 = lda(train_M015, train_l015)            # train LDA 3 digits
LD0157 = lda(train_M0157, train_l0157)         # train LDA 4 digits
LDall = lda(Mtrain, ltrain)                    # train all 10 digits
pred12lda  = LD12.predict(val_M12)             # PREDICT (classify, 2)
pred1015da = LD015.predict(val_M015)           # PREDICT (classify, 3)
predAllLda = LDall.predict(Mtest)              # PREDICT (classify, all 10)
F3l, fLab3l = confusion(pred1015da, val_l015) # Analyze error/misclassification
F10l, fLab10l = confusion(predAllLda, ltest)  # Analyze error/misclassification

#==== SVM ====
SVM12 =  svm(train_M12,  train_l12)            # train SVM 2 digits
SVM015 = svm(train_M015, train_l015)           # train SVM 3 digits
SVM = svm(Mtrain, ltrain)                      # train all 10 digits
predl12svm = SVM12.predict(val_M12)
predl015svm = SVM015.predict(val_M015)
predAllSvm = SVM.predict(Mtest)
F10s, fLab10s = confusion(predAllSvm, ltest)  # Analyze error/misclassification

#==== DTREE ====
DT12 = dtree(train_M12,  train_l12)
pred12dtree = DT12.predict(val_M12)

#==== output ====
plotSV(S)
plotImageMatrix([Ured[:,0],Ured[:,1],Ured[:,2],Ured[:,3],Ured[:,4],Ured[:,5]],2,3) # first 6 eigendigits
plot3(P[:,0][0:1000], P[:,1][0:1000], P[:,2][0:1000], c=ltrain[0:1000], ttl='All Digits projected on PC 1,2,3', cbar=True, fname='')
plot3(P[:,1][0:1000], P[:,2][0:1000], P[:,3][0:1000], c=ltrain[0:1000], ttl='All Digits projected on PC 2,3,4', cbar=True, fname='')
```

```
Rs = np.matmul(train_M12[0:12], Ured.T) #- from PCA compressed space back to original dimensions
plotImageMatrix(Rs,3,4)
Rs = np.matmul(train_M89[0:12], Ured.T) #- from PCA compressed space back to original dimensions
plotImageMatrix(Rs,3,4)
RsAll = np.matmul(P, Ured.T)
plotImageMatrix(RsAll[0:12],3,4)
plotLDA(LD01.transform(train_M01), train_l01, ttl='LDA of digits 0 & 1')
plotLDA(LD89.transform(train_M89), train_l12, ttl='LDA of digits 8 & 9')
plotLDA(LD015.transform(train_M015), train_l015, ttl='LDA of digits 0,1,5')
plotLDA(LD0157.transform(train_M0157), train_l0157, ttl='LDA of digits 0,1,5,7')
print('train 1,2', train_M12.shape, 'val 1,2', val_M12.shape)
print('2 digit prediction')
for k in range(10): print('\tpred LDA', pred12lda[k],  'pred SVM', predl12svm[k], 'D Tree', pred12dtree[k], 'true', val_l12[k])
print('3 digit prediction')
for k in range(20): print('\tpred LDA', pred1015da[k], 'pred SVM', predl015svm[k], 'true', val_l015[k])
print('10 digit prediction')
for k in range(20): print('\tpred LDA', predAllLda[k], 'pred SVM', predAllSvm[k], 'true', ltest[k])

print('---- 3 digits LDA ----')
print('label indices', fLab3l)
for r in F3l:
        print(r)
correct = np.trace(F3l)
wrong = np.sum(F3l) - correct
print('Correct', correct, 'wrong', wrong)

print('---- 10 digits LDA ----')
correct = np.trace(F10l)
wrong = np.sum(F10l) - correct
for r in F10l:
        print(r)
print('Correct', correct, 'wrong', wrong)

print('---- 10 digits SVM ----')
correct = np.trace(F10s)
wrong = np.sum(F10s) - correct
for r in F10s:
        print(r)
print('Correct', correct, 'wrong', wrong)
```

Pair-wise digit separation performance *allpairs.py* as described in Section III:

```
import pickle, numpy as np
import matplotlib.pyplot as plt
from tensorFiles import *
from pcaF import *
from plottingFunctions import *
from classifier import *
#=== MNIST data ===
M1,Mtest = tnsrFile2numpy('data/mnist/mnist_train_images.npz'), tnsrFile2numpy('data/mnist/mnist_test_images.npz')
M3,L1 = tnsrFile2numpy('data/mnist/mnist_validate_images.npz'), tnsrFile2numpy('data/mnist/mnist_train_labels.npz')
Ltest,L3 = tnsrFile2numpy('data/mnist/mnist_test_labels.npz'),    tnsrFile2numpy('data/mnist/mnist_validate_labels.npz')
Mtrain, Ltrain = np.array(list(M1) + list(M3)), np.array(list(L1)+list(L3)) # 60K training images
ltrain,ltest = [],[]
for vbin in Ltrain: ltrain.append(np.argmax(vbin))
for vbin in Ltest: ltest.append(np.argmax(vbin))
print(Mtrain.shape, Mtest.shape)

#=== PCA ===
if True:
        nEv = 100
        Ured,S,P = pca(Mtrain, nv=nEv)
        Mtrain = np.matmul(Mtrain, Ured)
        Mtest = np.matmul(Mtest, Ured)
        print('TRAIN matrix after projection ', Mtrain.shape)

#=== pairwise ===
pairs = allPairs(10)
E,Etr = {},{}
for SEL in [3] : #1,2,3]:
        try: E[SEL]
        except: E[SEL] = []
        try: Etr[SEL]
        except: Etr[SEL] = []
        for pair in pairs:
                train_M, train_l   = selectDigits(pair, Mtrain, ltrain)#, sz=sz)
                [train_M, train_l] = randperm([train_M, train_l])
                val_M, val_l = selectDigits(pair,   Mtest, ltest)#, sz=sz)
                if   SEL == 1: Model = lda(train_M,  train_l)
                elif SEL == 2: Model = svm(train_M,  train_l)
                elif SEL == 3: Model = dtree(train_M,  train_l, mxDepth=15)

                predict = Model.predict(train_M)
                nrErrTrain = len(np.nonzero(train_l - predict)[0])
                predict = Model.predict(val_M)
                nrErrEval = len(np.nonzero(val_l - predict)[0])

                E[SEL].append(  [pair, nrErrEval, len(val_l), nrErrTrain, len(train_l) ] )
                print(SEL, '** PAIR =', pair, 'EVAL err', nrErrEval,len(val_l), 'Train error', nrErrTrain, len(train_l))

#=== output ===
for e in E:
        for k in E[e] : print(e, k)
f = open('data/pairwiseError.dict', "wb")
pickle.dump(E,f)
f.close()
```

Basic statistics *pairStats.py*. This file processes output dictionary *pairwiseError.dict* generated by the *allpairs.py* script. This script produces the error percent table output (Tables 1, 2 and 3)

```
import pickle, numpy as np
E = pickle.load(open('data/pairwiseError.dict','rb'))
M1,M2,M3 = np.zeros((10,10)), np.zeros((10,10)), np.zeros((10,10))
tbest1,tbest2,tbest3,tworst1,tworst2,tworst3 = 999,999,999,0,0,0
tb1,tb2,tb3,tw1,tw2,tw3 = None,None,None,None,None,None
for m in E:
        for k in E[m]:
                eEv, eTr = (k[1]/k[2])*100.0, (k[3]/k[4])*100.0
                print(m, k[0], eEv, eTr)
                if m == 1:
                        M1[ k[0][0],k[0][1] ] = eEv
                        if eTr > tworst1:
                                tworst1 = eTr
                                tw1 = k[0]
                        if eTr < tbest1:
                                tbest1 = eTr
                                tb1 = k[0]
                elif m==2:
                        M2[ k[0][0],k[0][1] ] = eEv
                        if eTr > tworst2:
                                tworst2 = eTr
                                tw2 = k[0]
                        if eTr < tbest2:
                                tbest2 = eTr
                                tb2 = k[0]
                elif m==3:
                        M3[ k[0][0],k[0][1] ] = eEv
                        if eTr > tworst3:
                                tworst3 = eTr
                                tw3 = k[0]
                        if eTr < tbest3:
                                tbest3 = eTr
                                tb3 = k[0]
print('LDA best train', tb1, tbest1, 'SVM best train', tb2, tbest2, 'CTree best train', tb3, tbest3)
print('LDA worst train', tw1, tworst1, 'SVM worst train', tw2, tworst2, 'CTree worst train', tw3, tworst3)
print(np.sum(M1), np.sum(M2), np.sum(M3))
print('--------  1  -------')
for n, row in enumerate(M1):
        szTex = "%d & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f \\\\" % (n,\
                row[1],row[2],row[3],row[4],row[5],row[6],row[7],row[8],row[9])
        print(szTex)
print('--------  2  -------')
for n, row in enumerate(M2):
        szTex = "%d & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f \\\\" % (n,\
                row[1],row[2],row[3],row[4],row[5],row[6],row[7],row[8],row[9])
        print(szTex)
print('--------  3  -------')
for n, row in enumerate(M3):
        szTex = "%d & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f & %3.1f \\\\" % (n,\
                row[1],row[2],row[3],row[4],row[5],row[6],row[7],row[8],row[9])
        print(szTex)
```

C. Additional information: variance/covariance of vectors of random variables

Expectation & variance/covariance of a vector random variable
Consider a vector random variable $\vec{X}$. This is really a collection of n random variables which we can organize into a vector $\vec{X} = (X_1\ X_2\ X_3 \dots)^T$. Then cumulative distribution function (CDF) of the vector random variable:

$$\begin{aligned} F_{\vec{X}}(\vec{x}) &= F_{X_1,X_2,X_3,\dots}(x_1, x_2, \dots, x_n) && (11)\\ &= P(X_1 < x_1,\ X_2 < x_2, \dots,\ X_n < x_n) && (12) \end{aligned}$$

So the cumulative distribution function is a function of vector position $\vec{x} \in \mathbb{R}^n$. The expected value of the random vector:

$$E\vec{X} = \begin{pmatrix} EX_1 \\ EX_2 \\ \vdots \\ EX_n \end{pmatrix} \tag{13}$$

Note: this expectation is not a scalar but the elements EX_i are traditional scalar expectations.
The variance/covariance matrix of *one* random vector $\vec{X}$ is the outer product:

$$\begin{aligned} C_{\vec{X}} &= E[(\vec{X} - E\vec{X})(\vec{X} - E\vec{X})^T] \qquad (i) && (14)\\ &= E\left[\begin{pmatrix} (X_1 - EX_1)^2 & (X_1 - EX_1)(X_2 - EX_2) & \dots & (X_1 - EX_1)(X_n - EX_n) \\ (X_2 - EX_2)(X_1 - EX_1) & (X_1 - EX_1)^2 & \dots & (X_2 - EX_2)(X_n - EX_n) \\ \vdots & & & \\ (X_n - EX_n)(X_1 - EX_1) & (X_n - EX_n)(X_2 - EX_2) & \dots & (X_1 - EX_1)^2 \end{pmatrix}\right] && (15)\\ &= \begin{pmatrix} Var(X_1) & Cov(X_1, X_2) & \dots \\ \vdots & & \end{pmatrix} && (16) \end{aligned}$$

In $\vec{X} - E\vec{X}$ both terms are vectors (see note above). Then (i) By definition of matrix product: $E[(\vec{X} - E\vec{X})(\vec{X} - E\vec{X})^T] \implies C_{ij} = (X_i - EX_i)(X_j - EX_j)$ so elements C_{ij} of $C_{\vec{X}}$ are covariance of random vector (scalar) elements $cov(X_i, X_j)$ and diagonal C_{ii} is variance $var(X_i)$. This is the variance/covariance matrix of a single vector random variable. Now what is the cross-variance/cov of *two* random vectors? First consider two vector random variables $\vec{X}$ and $\vec{Y}$. Now we get the **Cross-covariance matrix** of two different random vectors.

$$\begin{aligned} C_{\vec{X}\vec{Y}} &= E[(\vec{X} - E\vec{X})(\vec{Y} - E\vec{Y})^T] && (17)\\ &= E[\vec{X}\vec{Y}^T] - E\vec{X}(E\vec{Y})^T && (18)\\ &= E\left[\begin{pmatrix} (X_1 - EX_1)(Y_1 - EY_1) & (X_1 - EX_1)(Y_2 - EY_2) & \dots & (X_1 - EX_1)(Y_n - EY_n) \\ (X_2 - EX_2)(Y_1 - EY_1) & (X_2 - EX_2)(Y_2 - EY_2) & \dots & (X_2 - EX_2)(Y_n - EY_n) \\ \vdots & & & \\ (X_n - EX_n)(Y_1 - EY_1) & (X_n - EX_n)(Y_2 - EY_2) & \dots & (X_n - EX_n)(Y_n - EY_n) \end{pmatrix}\right] && (19) \end{aligned}$$

The i, j entry is the scalar $(X_i - EX_i)(Y_j - EY_j) = cov(X_i, Y_j)$ of scalar random variable elements X_i and Y_j.

Example: random vector with 2 elements and 3 measurements
If $\vec{X}$ is an image, it has only 2 pixels. Each pixel is a random variable (element) in the vector $\vec{X}$ so that $\vec{X} = (X_1\ X_2)^T$

Variance-cov matrix

$$
\begin{aligned}
Cov(\vec{X}) &= E[(\vec{X} - E\vec{X})(\vec{X} - E\vec{X})^T] && (20)\\
&= E\left[\left(\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} - \begin{pmatrix} \mu_{X_1} \\ \mu_{X_2} \end{pmatrix}\right)\left(\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} - \begin{pmatrix} \mu_{X_1} \\ \mu_{X_2} \end{pmatrix}\right)^T\right] && (21)\\
&= E\left[\begin{pmatrix} X_1 - \mu_{X_1} \\ X_2 - \mu_{X_2} \end{pmatrix}\begin{pmatrix} X_1 - \mu_{X_1} \\ X_2 - \mu_{X_2} \end{pmatrix}^T\right] && (22)\\
&= E\left[\begin{pmatrix} (X_1 - EX_1)(X_1 - EX_1) & (X_1 - EX_1)(X_2 - EX_2) \\ (X_2 - EX_2)(X_1 - EX_1) & (X_2 - EX_2)(X_2 - EX_2) \end{pmatrix}\right] && (23)\\
&= E\left[\begin{pmatrix} (X_1 - \mu_{X_1})(X_1 - \mu_{X_1}) & (X_1 - \mu_{X_1})(X_2 - \mu_{X_2}) \\ (X_2 - \mu_{X_2})(X_1 - \mu_{X_1}) & (X_2 - \mu_{X_2})(X_2 - \mu_{X_2}) \end{pmatrix}\right] && (24)\\
&= \frac{1}{?} A_c A_c^T && (25)
\end{aligned}
$$

Note that in the above matrix, all values are scalars. We can get the means μ_{X_1} and μ_{X_2} from our 3 measurements. Our data looks like:

$$
A = \begin{pmatrix} X_{1_1} & X_{1_2} & X_{1_3} \\ X_{2_1} & X_{2_2} & X_{2_3} \end{pmatrix}, \quad A_c = A - M_\mu = \begin{pmatrix} X_{1_1} - \mu_{X_1} & X_{1_2} - \mu_{X_1} & X_{1_3} - \mu_{X_1} \\ X_{2_1} - \mu_{X_2} & X_{2_2} - \mu_{X_2} & X_{2_3} - \mu_{X_2} \end{pmatrix},
$$

where X_{1_1} is scalar random variable (random vector *element*) X_1 measured at one time (or object instance). One photo would be an instance. Note there are 2 total random variables organized into a single random vector $\vec{X}$. There are 3 image instances. We have expectation $EX_i = \mu$ of each random variable pixel by taking the average of the samples. **Now what are scalar X_1 and X_2?** We also note we take the *expectation* of the above matrix so in each element we have $E[(X_i - \mu_{X_i})(X_j - \mu_{X_j})]$. Remember expectation (discrete distribution) $EZ = \sum_k p_k z_k$. So for the top left element if each way equal:

$$
E[(X_1 - \mu_{X_1})(X_1 - \mu_{X_1})] = 1/3[(X_{1_1} - \mu_{X_1})(X_{1_1} - \mu_{X_1}) + (X_{1_2} - \mu_{X_2})(X_{1_2} - \mu_{X_2}) + (X_{1_3} - \mu_{X_3})(X_{1_3} - \mu_{X_3})]
$$

MNIST random vectors & PCA
MNIST PCA is like the above example but with 784 pixels: there is a single random vector $\vec{X}$ that contains 784 pixels (784 random variables). All the 60000 images (rows in A) are samples or object instances analogous to time points. So the variance-covariance matrix of $\vec{X}$ is 784×784

D. Additional information: LDA, SVM & Ctree error analysis

With PCA compression into 100 features. Confusion matrix generated by *hw4.py* for 10 digits LDA:

$$M_C = \begin{pmatrix} 927 & 0 & 14 & 5 & 0 & 13 & 10 & 2 & 6 & 9 \\ 0 & 1089 & 37 & 6 & 10 & 9 & 7 & 31 & 28 & 7 \\ 3 & 3 & 835 & 25 & 5 & 3 & 5 & 18 & 7 & 3 \\ 3 & 2 & 23 & 877 & 1 & 52 & 0 & 6 & 31 & 11 \\ 0 & 1 & 20 & 3 & 877 & 12 & 19 & 15 & 16 & 45 \\ 21 & 3 & 8 & 37 & 4 & 717 & 29 & 1 & 40 & 6 \\ 13 & 4 & 16 & 4 & 12 & 16 & 880 & 2 & 13 & 3 \\ 2 & 2 & 11 & 19 & 1 & 16 & 0 & 873 & 9 & 14 \\ 9 & 31 & 54 & 24 & 7 & 39 & 8 & 4 & 797 & 12 \\ 2 & 0 & 14 & 10 & 65 & 15 & 0 & 76 & 27 & 899 \end{pmatrix}$$

Total correct $Trace(M_C) = 8771$, total wrong 1229.

Confusion matrix generated by *hw4.py* for 10 digits SVM:

$$M_C = \begin{pmatrix} 966 & 0 & 7 & 4 & 1 & 10 & 7 & 1 & 5 & 6 \\ 0 & 1124 & 2 & 1 & 1 & 3 & 1 & 7 & 3 & 7 \\ 0 & 3 & 977 & 17 & 7 & 3 & 9 & 20 & 6 & 3 \\ 3 & 1 & 7 & 946 & 0 & 35 & 1 & 4 & 26 & 10 \\ 0 & 0 & 10 & 0 & 939 & 5 & 4 & 7 & 8 & 33 \\ 5 & 1 & 4 & 17 & 0 & 801 & 13 & 0 & 30 & 6 \\ 3 & 2 & 6 & 0 & 6 & 12 & 921 & 0 & 9 & 0 \\ 2 & 1 & 7 & 9 & 2 & 0 & 0 & 970 & 4 & 20 \\ 1 & 3 & 12 & 15 & 4 & 21 & 2 & 2 & 880 & 5 \\ 0 & 0 & 0 & 1 & 22 & 2 & 0 & 17 & 3 & 919 \end{pmatrix}$$

Total correct $Trace(M_C) = 9443$, total wrong 557.