MPr701 Project
Report

Handwritten Digit Recognition using python

Submitted by
Aron G.
M013704

Under the guidance of
Dr. Ajit Kumar
Institute of Chemical Technology, Mumbai

Centre for Excellence in Basic Sciences
University of Mumbai
Kalina, Mumbai, India – 400098

Autumn Semester 2016

Abstract

The ability of computers to classify things on its own is a stepping stone for unsupervised machine learning, in particular, artificial intelligence. This is no different from being able to recognize and classify a new pattern by learning from a pre-classified training data. The aim of this project is to localize the training data set to handwritten digits and to analyze how the machine is able to recognize an unknown digit, by studying some of the mainstream methods used in classification and writing python programs for a few.

Contents

1 Introduction

1.1 Multiclass classification

In machine learning, multiclass or multinomial classification is the problem of classifying instances into one of the more than two classes. While some of them permit the use of multiple classes simultaneously, others use multiple binary classification. In this project, we use the latter. The reason being, binary classification is easier and the variations between classes can be analyzed better.
In this project, the classes will have handwritten digits as training samples. Since there are ten digits $(0-9)$, there are ten classes, named as digit classes. Each class has training images of a particular digit, and no other class will have it. The aim is to build a classifier using these ten digit classes and to identify an unknown digit sample by placing it in the right class.

1.2 The Database

The database used throughout the project is the standard MNIST (*Mixed National Institute of Standards and Technology database*) of handwritten digits. The database contains 60,000 training images(6000 each for 10 different classes of digits from 0 to 9) and 10,000 testing images. Each image is 28×28 pixels. Sample images from each of the ten digit classes are shown in Figure 1. Also, another dataset called USPS is mentioned in the project. The

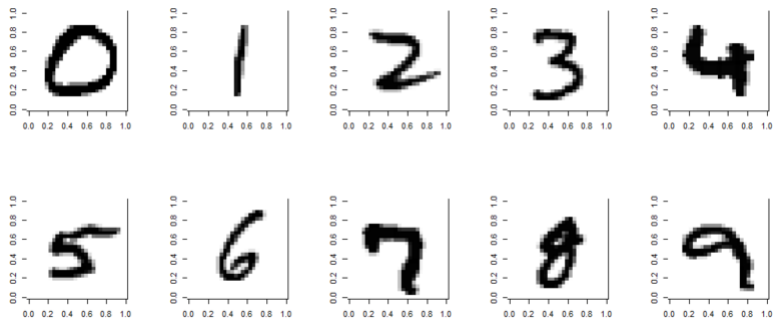

Figure 1: Digit samples from the database.[5]

dataset refers to numeric data obtained from the scanning of handwritten digits from envelopes by the U.S. Postal Service. The original scanned digits are binary and of different sizes and orientations. The images here have been deslanted and size normalized, resulting in 16 x 16 gray scale images. There

are 7291 training samples and 2007 test samples in this database. Though they are not used in the project, they are referred to in table[3] to show the performance comparison of a few methods.

2 Methods of classification

There are various methods for classifying patterns, the ones that are explored are the following:

1. K-Nearest Neighbours
2. Support Vector Machines
3. SVD bases classification
4. Tangent method classification

2.1 K-Nearest Neighbours

K-Nearest Neighbors(or KNN in short) is a non-parametric method, which is one of the most primitive methods used for pattern recognition. The reason being the simplicity of the method.
The digits are treated in three but equivalent forms;

1. As 28×28 gray scale images.
2. As functions of two variables, $s = s(x, y)$.[Figure 2]
3. As vectors in $\mathbb{R}^{784}$.

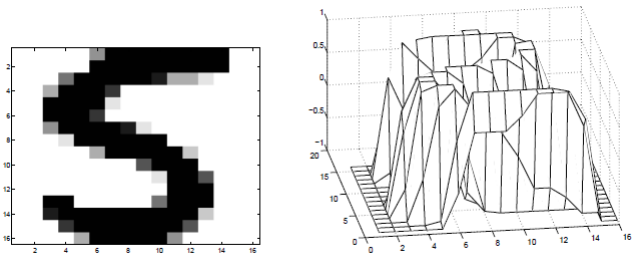

Figure 2: Digits as functions to two variables.[2]

Euclidean distance is used to measure the distance here. Stacking the columns of the image matrix one over the other results in making the images as vectors in $\mathbb{R}^{784}$. Once done so, define the distance function as

$$d(x, y) = \|x - y\|_2 ,$$

where x and y are vectors in $\mathbb{R}^{784}$. Once the distance function is established, the next step is to compute the means(*centroids*) of images of each digit class. Here, there are ten classes$(0 - 9)$, and hence ten *centroids* images [Figure 3]. Now calculate the distance of the unknown image from each of the ten *centroids* images, and assign the unknown to the one with the least distance.

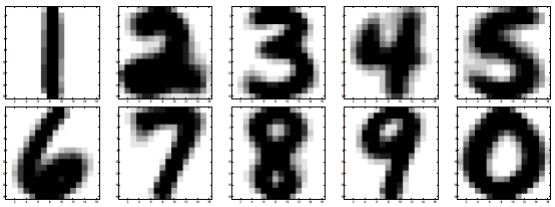

Figure 3: The means(*centroids*) of digits in the training set.[2]

2.2 Support Vector Machines

Support Vector Machines(SVM) are supervised learning methods used for analyzing data for classification and regression.
Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other. This make s it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

2.2.1 How it works

A support vector machine constructs a hyperplane or set of hyperplanes in a high-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class

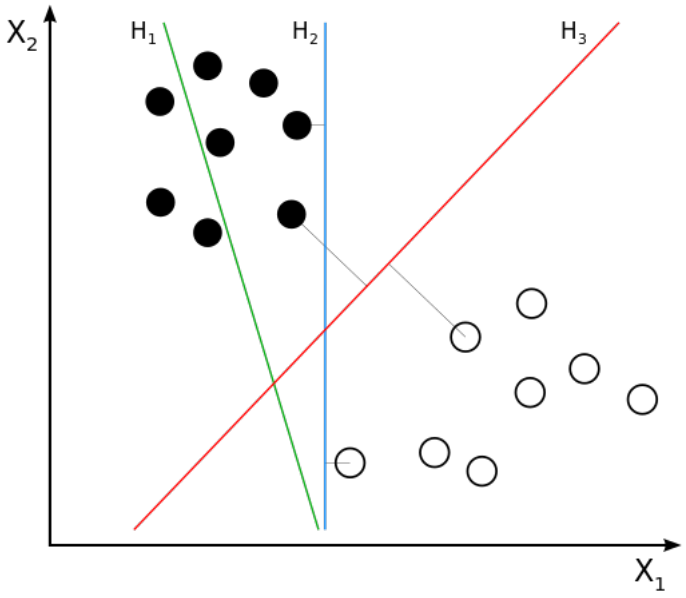

Figure 4: Graphic showing how a support vector machine would choose a separating hyperplane for two classes of points in 2D. H_1 does not separate the classes. H_2 does, but only with a small margin. H_3 separates them with the maximum margin.[4]

(so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier [Figure 4].

2.2.2 Linear SVM

Given n points in the training set

$$\{(\vec{x_1}, y_1), (\vec{x_2}, y_2), \ldots, (\vec{x_n}, y_n)\}$$

where $\vec{x_i} \in \mathbb{R}^p$ and $y_i \in \{+1, -1\}$. The aim is to find the maximum margin hyperplane that divides the points $\vec{x_i}$'s for which $y_i = +1$ from the points for which $y_i = -1$. Also the distance between the hyperplane and the nearest point $\vec{x_i}$ from either group should be maximized. Any hyperplane satisfied by the the set of points $\vec{x_i}$ can be expressed as,

$$\vec{w} \cdot \vec{x} - b = 0 \tag{1}$$

where $\vec{w}$ is the normal vector to the hyperplane. The parameter $\frac{b}{\|\vec{w}\|}$ determines the offset of the hyperplane from the origin along the normal vector $\vec{w}$.
Since the training data is linearly separable, two parallel hyperplanes can be

placed in between them in such a way that the distance between them is maximum.
Their median is called the margin. The parallel margins can be described by the equations

$$\vec{w} \cdot \vec{x} - b = 1 \quad and \quad \vec{w} \cdot \vec{x} - b = -1 \tag{2}$$

making the distance between them, $\frac{2}{\|\vec{w}\|}$. Maximizing this distance is the same as minimizing $\|\vec{w}\|$. The following constraint is added to equation(2) to prevent data from falling into the margin

$$\vec{w} \cdot \vec{x_i} - b \begin{cases} \geq 1 & y_i = 1 \\ \leq -1 & y_i = -1 \end{cases}$$

This is the same as writing

$$y_i(\vec{w} \cdot \vec{x_i} - b) \geq 1, \forall i \in \{1, 2, \ldots, n\} \tag{3}$$

Now, the problem has been reduced to a constraint based optimization which can be solved using Legrange multipliers, that is, to minimize $\|\vec{w}\|$ subject to $y_i(\vec{w} \cdot \vec{x_i} - b) \geq 1$, for $i = 1, \ldots, n$
The $\vec{w}$ and b that satisfies the above determines the classifier, $\vec{z} \mapsto sgn(\vec{w} \cdot \vec{x} - b)$, where $\vec{z}$ is the unclassified vector. Geometrically, the maximum margin hyper plane is completely determined by those $\vec{x_i}$ which lie nearest to it, which are called as support vectors [Figure 5].

2.2.3 Nonlinear SVM

The motivation towards building a nonlinear classifier is that, after transforming the input space into a higher dimensional feature space, they can be linearly classified [Figure 6]. A *kernel trick* is used to bypass the procedure of finding the transformation explicitly, which is otherwise computationally expensive.

Since any linear transformation can be expressed as a dot product, all the dot products in the classifier is replaced by a kernel function. Let ϕ be the transformation on $\vec{x_i}$, then the kernel associated with the transformation is given by

$$k(\vec{x_i}, \vec{x_j}) = \phi(\vec{x_i}) \cdot \phi(\vec{x_j})$$

Some of the most popular kernel functions are:

1. Polynomial Kernel : $k(\vec{x_i}, \vec{x_j}) = (\vec{x_i} \cdot \vec{x_j})^d$.

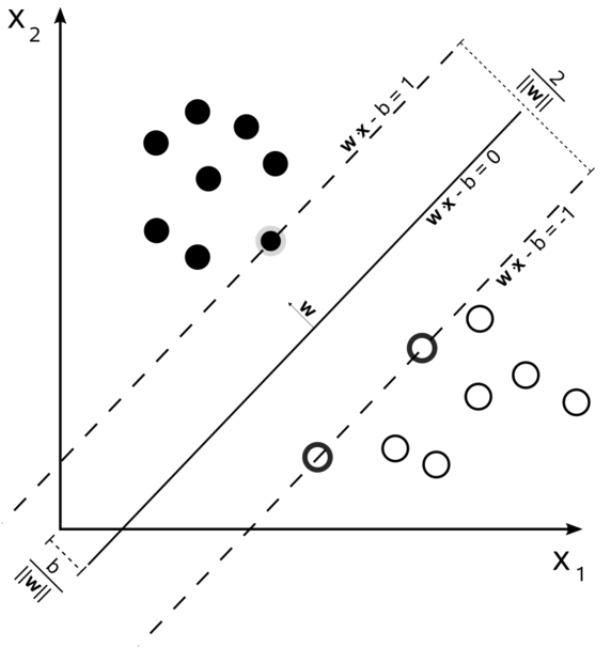

Figure 5: Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors.[4]

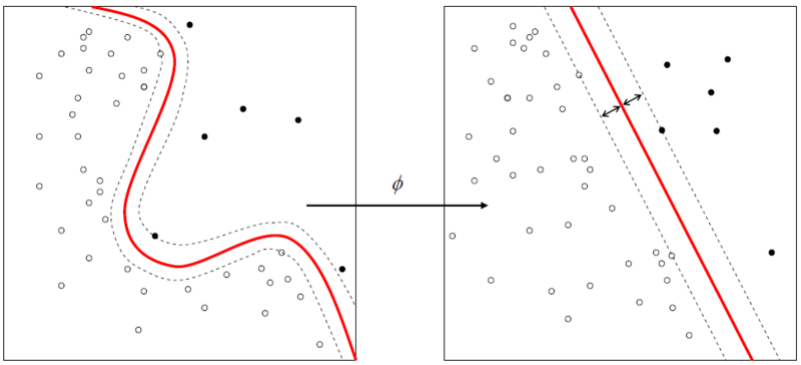

Figure 6: Transformation from input space into feature space.[4]

2. Round Basis Function (RBF Kernel) : $k(\vec{x_i}, \vec{x_j}) = e\left(\frac{\|\vec{x_i} - \vec{x_j}\|^2}{2\sigma^2}\right)$.

3. Hyperbolic Tangent Kernel : $k(\vec{x_i}, \vec{x_j}) = tanh(k\vec{x_i} \cdot \vec{x_j} + c)$ for some $k > 0$ and $c < 0$.

2.2.4 Building the classifier

To extend SVM to cases in which the data are not linearly separable, we introduce the *hinge loss function*,

$$max\{0, 1 - y_i(\vec{w}\vec{x_i} - b)\}$$

The function is zero if the constraint in (3) is satisfied, that is, if $\vec{x_i}$ lies on the correct side of the margin, if not, the function's value is proportional to the distance from the margin.
Now the aim is to minimize,

$$\left[\frac{1}{n}\sum_{i=1}^{n} max\{0, 1 - y_i(\vec{w} \cdot \vec{x_i} - b)\}\right] + \lambda \left\|\vec{w}\right\|^2 \tag{4}$$

where the parameter λ determines the trade off between increasing the margin-size and ensuring that the $\vec{x_i}$ lie on the correct side of the margin. Minimizing the above can be seen as a constraint based optimization problem with a differentiable objective function as defined in the following way, for $i \in \{1, 2, \ldots, n\}$, define $\zeta_i = max\{0, 1 - y_i(w \cdot x_i + b)\}$. Then, (4) reduces to the *primal problem*

$$minimize \quad \frac{1}{n}\sum_{i=1}^{n} \zeta_i + \lambda \left\|w\right\|^2 \tag{5}$$

$$\text{subject to } y_i(w \cdot x_i + b) \geq 1 - \zeta_i \text{ and } \zeta_i \geq 0, \forall i$$

Solving the Lagrangian dual of (5), it simplifies to

$$maximize \quad f(c_1, \ldots, c_n) = \sum_{i=1}^{n} c_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i c_i (\vec{x_i} \cdot \vec{x_j}) y_j c_j \tag{6}$$

$$\text{subject to } \textstyle\sum_{i=1}^{n} c_i y_i = 0, \text{ and } 0 \leq c_i \leq \frac{1}{2n\lambda}, \forall i$$

The problem(6) is called the dual problem. Since the dual maximization problem is a quadratic function of the c_i subject to linear constraints, it is efficiently solvable by quadratic programming algorithms. Here, variables c_i's are defined by the equation,

$$\vec{w} = \sum_{i=1}^{n} c_i y_i \vec{x_i}$$

Moreover, $c_i = 0$ exactly when x_i lies on the correct side of the margin, and $0 \leq c_i \leq (2n\lambda)^{-1}$ when $\vec{x_i}$ lies on the margin's boundary. It follows that $\vec{w}$

can be written as a linear combination of support vectors.
The offset, b, can be recovered by finding an $\vec{x_i}$ on the margin's boundary and solving

$$y_i(\vec{w_i} \cdot \vec{x_i} + b) = 1$$

As mentioned earlier, we need to make use of the kernel trick to transform data points to a feature space, this is achieved by modifying (6) as

$$maximize \quad f(c_1, \ldots, c_n) = \sum_{i=1}^{n} c_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i c_i (\phi(\vec{x_i}) \cdot \phi(\vec{x_j})) y_j c_j \qquad (7)$$

and given a kernel function k, that satisfies $k(\vec{x_i}, \vec{x_j}) = \phi(\vec{x_i}) \cdot \phi(\vec{x_j})$, then, equation(7) can be written as

$$maximize \quad f(c_1, \ldots, c_n) = \sum_{i=1}^{n} c_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i c_i k(\vec{x_i}, \vec{x_j}) y_j c_j$$

$$\text{subject to } \textstyle\sum_{i=1}^{n} c_i y_i = 0, \text{ and } 0 \leq c_i \leq \frac{1}{2n\lambda}, \forall i$$

The coefficients c_i can be solved for using quadratic programming, as before. Again, we can find some index i such that $0 \leq c_i \leq \frac{1}{2n\lambda}$, so that $\phi(x_i)$ lies on the boundary of the margin in the transformed space, and then solve,

$$\begin{aligned} b \quad = \vec{w} \cdot \phi(\vec{x_i}) - y_i \quad &= \left[\sum_{k=1}^{n} c_k y_k \phi(\vec{x_k}) \cdot \phi(\vec{x_i}) \right] - y_i \\ &= \left[\sum_{k=1}^{n} c_k y_k k(\vec{x_k}, \vec{x_i}) \right] - y_i \end{aligned}$$

Finally the new points can be classified by computing,

$$\vec{z} \mapsto sgn(\vec{w} \cdot \phi(\vec{z}) + b) = sgn\left(\left[\sum_{i=1}^{n} c_i y_i k(\vec{x_i}, \vec{z}) \right] + b \right)$$

2.3 One v/s Rest strategy

The *one v/s rest* strategy involves training a single classifier per class, with the samples of that class as positive samples and all other samples as negatives.
Essentially, it is a multiple binary classifier. In our case, we have ten digit classes and hence ten classifiers, each for one class.
Let $\{C_i, i \in \{0, \ldots, 9\}\}$ be the set of classifiers and let x be an unknown

digit. Each classifier will determine how close x is with a digit class i with a certain confidence f_i. And finally, x belongs to the image class which has the maximum confidence. That is,

$$x \in \{i | f_i = max\{f_0, \ldots, f_9\}\}$$

2.4 One v/s One strategy

The *one v/s one* classifier constructs one classifier per pair of classes. Since there are $n(n-1)/2$ pairs, there are $n(n-1)/2$ classifiers. Here, the value of n is ten, so there are forty-five classifiers.$\forall i, j \in \{0, \ldots, 9\}, i \neq j$, define $C_{i,j} = C_{j,i}$ =binary classifier between the digit classes i and j. The steps in the strategy as as follows.

- Let x be an unknown digit sample.
- Then, x is fitted using all the forty-five $C_{i,j}$'s.
- For each $C_{i,j}$, either i or j wins with a certain confidence.
- The class which received the most votes is selected as the class to which x belongs.
- In the event of a tie (among two classes with an equal number of votes), it selects the class with the highest aggregate classification confidence by summing over the pair-wise classification confidence levels computed by the underlying binary classifiers.

Since it requires to fit $n(n-1)/2$ classifiers, this is slower than *one v/s rest* strategy.

2.5 Classification using SVD Bases

2.5.1 Singular Value Decomposition(SVD)

Suppose M is aa $m \times n$ matrix, who has real or complex entries. Then there exists a factorization, called a singular value decomposition of M, of the form

$$M = U\Sigma V^*,$$

where, U ia an $m \times m$, unitary matrix.
Σ is a diagonal $m \times n$ matrix with non-negative real numbers on the diagonal.
V^* is an $n \times n$, unitary matrix.

2.5.2 Method

The images, as we recall are 28×28 matrices, and stacking columns on top of each other makes each image a vector in $\mathbb{R}^{784}$. Let $A \in \mathbb{R}^{m \times n}$, with $m = 784$ and n =number of training digits of one of the ten different classes of digits [Figure 7]. The columns of A span a linear subspace of $\mathbb{R}^m$. However this

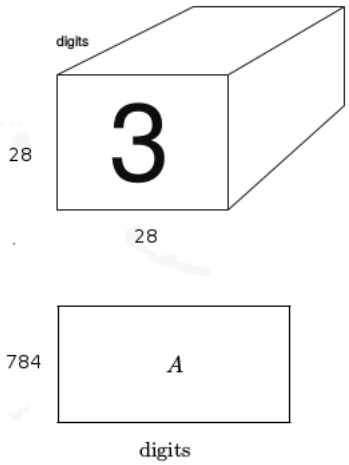

Figure 7: The image of one digit is a matrix, and the set of images of one kind form a tensor(above) and the matrix A(below).[2]

subspace cannot be expected to have a larger dimension, because if it did, then the subspaces of different kinds of digits would intersect. The aim is to model the variation within the set of training and test digits of one kind using an orthogonal basis of the subspace. This can be achieved using SVD.

and any matrix is a sum of rank 1 matrices, thus

$$A = \sum_{i=1}^{m} \sigma_i u_i v_i^T \tag{8}$$

Each column of A represents the image of a digit class and therefore the left singular vectors u_i are an orthogonal basis in the *image space* of that particular digit class. They are called *singular images* [Figure 8].

From (5), the j^{th} column of A is equal to

$$a_j = \sum_{i=1}^{m} (\sigma_i v_{ij}) u_i \tag{9}$$

Since the first singular vector contains the *dominating* direction of the data matrix, folding the column vectors u_i's back into 28×28 images, the first

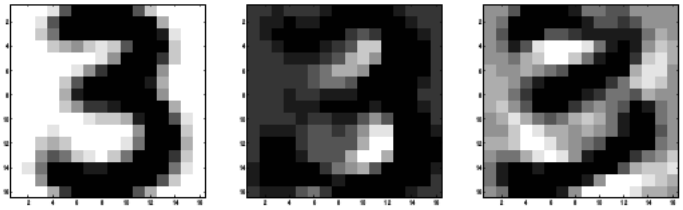

Figure 8: The first three singular images.[2]

one should look like the digit class we started with, and the following ones representing the dominating variations of the training set around the first singular image.
The basic assumptions regarding SVD basis classification are:

- Each digit in the training and test set is characterized by a few of the first singular values of its own kind.
- An expansion in terms of the first few singular images discriminates well between the different classes of digits.
- If an unknown digit can be better approximated in one particular basis of singular images of a certain class, than in the bases of the other classes, then it is likely that the unknown digit belongs to the former class.

Now we determine how well an unknown digit can be represented in 10 different bases, each belonging to subspace of their respective digit class. This is achieved by computing the residual vector in the least squares problems of the type

$$\min_{\alpha_i} \left\| z - \sum_{i=1}^{k} \alpha_i u_i \right\| \tag{10}$$

where z represents the unknown digit and u_i represents the singular images. An example residuals are shown in Figure 9.
The equation(10) can be rewritten as

$$\min_{\alpha_i} \|z - U_k \alpha\|_2 ,$$

where $U_k = (u_1, u_2, \ldots, u_k)$. Since the columns of U_k are orthogonal, the solution of the problem is given by $\alpha = U_k^T z$, and the norm of the residual

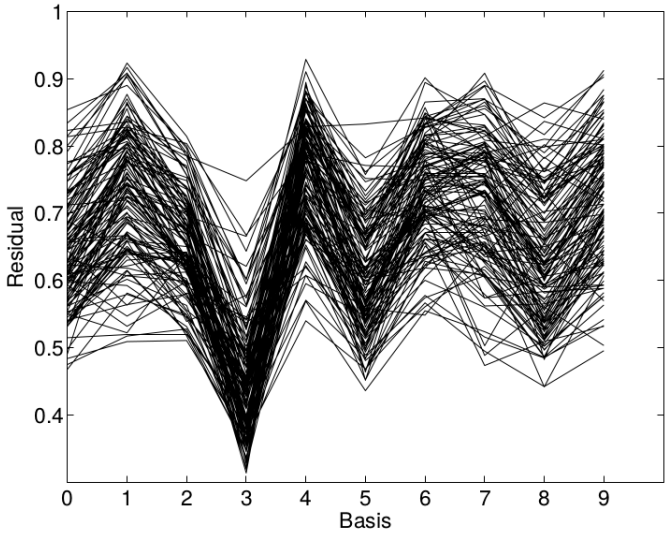

Figure 9: Relative residuals of all test 3's.[2]

vector of the least square problem is

$$\left\|(I - U_k U_k^T)z\right\|_2 \tag{11}$$

This can be visualized as the norm of the projection of the unknown digit onto the subspace orthogonal to span(U_k).

2.6 Tangent Distance Method

The limitations of the previous methods are that none of them are translation invariant, that is, the distance measure used(*Euclidean measure*) cannot accommodate deviations, therefore we need to device a new measure which is translation invariant. Hence, we take into account tangent distance as the distance measure, which is invariant under small transformations.
As we recall, the 28×28 images are points in $\mathbb{R}^{784}$. Let p be a fixed pattern in an image. Consider one transformation, say, translation along x-axis, then this transformation can visualized as a moving pattern along a curve in $\mathbb{R}^{784}$. Let the curve be parametrized by a real character α such that the curve is given by $s(p, \alpha)$ with $s(p, 0) = p$. In general, the curve is nonlinear and can be approximated by the first two terms in the Taylor expansion,

$$s(p, \alpha) = s(p, 0) + \frac{ds}{d\alpha}(p, 0) + O(\alpha^2) \approx p + t_p\alpha$$

where $t_p = \frac{ds}{d\alpha}(p, 0)$ is a vector in $\mathbb{R}^{784}$, with small perturbations of α around 0, a small movement of pattern along the tangent at point p is achieved. Consider another pattern with similar approximation,

$$s(e, \alpha) \approx e + t_e \alpha \tag{12}$$

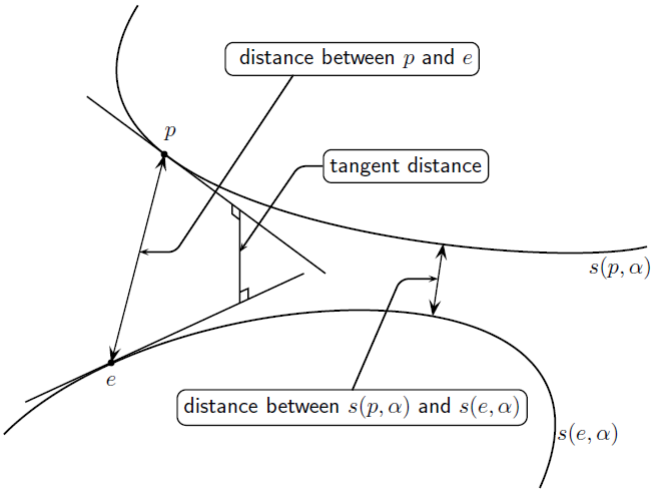

Figure 10: The distance between the points p and e, and the tangent distance.[2]

The distance function is not influenced much since the transformations made are very small. Hence the distance between p and e are defined to be the closest distance between the two curves. However, this is impossible to compute in general, hence we take the first order approximations and compute the closest distance between the two tangents in the points p and e [Figure 10]. Thus we move the patterns independently along their respective tangents, until the smallest distance is found. If we measure this distance in the usual Euclidean norm, we solve the least squares problem,

$$\min_{\alpha_p, \alpha_e} \|p + t_p \alpha_p - e - t_e \alpha_e\|_2 = \min_{\alpha_p, \alpha_e} \left\| (p - e) - \begin{pmatrix} -t_p & t_e \end{pmatrix} \begin{pmatrix} -\alpha_p \\ \alpha_e \end{pmatrix} \right\|_2 \tag{13}$$

Now, considering the case where the pattern p is moved in l different curves in $\mathbb{R}^{784}$, parametrized by $\alpha = \begin{pmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_l \end{pmatrix}^T$. This is equivalent to moving the pattern on an n-dimensional manifold in $\mathbb{R}^{784}$.

Now, going back to the problem, the tangent plane is given by the first two terms in the Taylor expansion of the function $s(p, \alpha)$.

$$s(p, \alpha) = s(p, 0) + \sum_{i=1}^{l} \frac{ds}{d\alpha_i}(p, 0)\alpha_i + O(\|\alpha\|_2^2) \approx p + T_p\alpha \tag{14}$$

where T_p is the matrix

$$T_p = \begin{bmatrix} \frac{ds}{d\alpha_1} & \frac{ds}{d\alpha_2} & \dots & \frac{ds}{d\alpha_l} \end{bmatrix}$$

and the derivatives are evaluated at the point $(p, 0)$.
Thus the tangent distance between the points p and e is defined as the smallest possible residual in the least squares problem,

$$\min_{\alpha_p, \alpha_e} \|p + T_p\alpha_p - e - T_e\alpha_e\|_2 = \min_{\alpha_p, \alpha_e} \left\| (p - e) - \begin{pmatrix} -T_p & T_e \end{pmatrix} \begin{pmatrix} -\alpha_p \\ \alpha_e \end{pmatrix} \right\|_2 \tag{15}$$

Now, think of p as pattern of a known digit, and e as pattern of an unknown digit, and since there are ten different classes of digits from where p can be chosen, the one to which e belongs depends on the least distance between e and the chosen p's.
The most important property of this distance function is that it is invariant under movements of the patterns on the tangent planes. For instance, if we make a small translation in the x-direction of a pattern, then with this measure, the distance it has been moved is equal to zero.

3 Results and Observations

3.1 K-Nearest Neighbours

The success rate of this algorithm is around 85%, which is not good enough. The reason for this relatively bad performance is that the algorithm does not use any information about the variation within each class of digits. The python program used to obtain the results are given at 5.1

3.2 Support Vector Machine

SVM proved to be a better classifier than K-NN, and it was the RBF kernel that could withstand most perturbations in the test image, simply because of it's flexibility. The time taken to generate the classifier is more in one v/s one than one v/s rest method but, the error rate is less in the former, leaving

the decision to the user in determining which strategy is superior according to their needs. The python program used to generate the classifier and the one used for recognition are given at 5.2.1 and 5.2.2 respectively.

3.2.1 One v/s Rest method

Classifier	**Error rate**	**Run time**(s)
SVM-Poly	1.77%	113
SVM-RBF	11.40%	25

Table 1: Performance of SVM using one vs. rest

3.2.2 One v/s One method

Classifier	**Error rate**	**Run time**(s)
SVM-Poly	1.52%	307
SVM-RBF	10.19%	62

Table 2: Performance of SVM using one vs. one

Performance sample

This is how the one vs. one classifier with a linear kernel performed on the image below.

3.3 SVD Bases Classification

A program for this method was not written, and the results are taken from an online source[6].An accuracy of 96.62% was noted in the paper and the database used was USPS and hence not to be compared with the results of the previous method, however it can be compared with table[3].

3.4 Tangent Distance method

The results for this is also taken from another source[1]. The database used for the below is USPS and hence not to be compared with the performance in MNIST database.

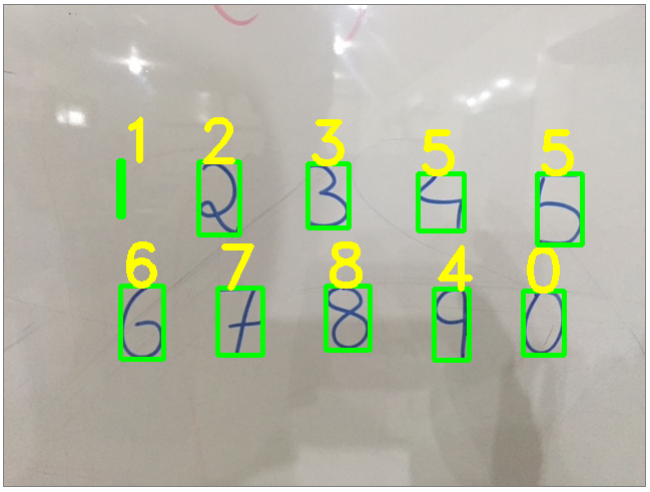

Figure 11: How the one vs one classifier with a linear kernel performs on the image, (*note how 4 and 9 are misclassified*).

Classifier	Accuracy
SVM-Poly	95.70%
SVM-RBF	95.42%
Tangent Distance	97.40%
Human performance	97.50%

Table 3: *Observe how close the performance of this method is to a human's* .[1]

4 Conclusion

The results prove that machines can be as good as humans in recognizing handwritten digits. Even though some of these techniques were familiar to us, optimizing them using regular mathematical tools is the key motivation behind the strategies used in the project. The factors that affect the prediction accuracy are the size of the training set, type of classifier and the variations in the training set. Tangent Distance Method was found to be the one with least errors and KNN with the most. In the SVM method, the one vs. one strategy produced better results than the one vs. rest, though the running time of the former was more than the latter.
The handwriting varies from person to person, so does the ambiguity, making it impossible for even humans to recognize with 100% accuracy. This makes handwritten digits an even platform to benchmark the accuracy of machine learning, by moderating the difficulty for humans and machines alike.

5 Python Programs

Python programs were written for KNN and SVM, using both one vs. one strategy and one vs. rest strategy. The LinearSVM module in python's scikit-image package follows the one vs. all rule and the SVM module follows the one vs. rest rule.

5.1 K-Nearest Neighbors

```
"""
Implementation of k-nearest neighbors on MNIST dataset.
"""

import numpy as np
from numpy import linalg as LA


'''
Returns array containing the k-nearest neighbors of point x
'''
def knn_x(data, x, k):
       n,m = data.shape

       # To calculate Euclidean distance efficiently,
       # use numpy subtraction/addition ops.
       x_data = np.tile(x,(n,1))
       dist_matrix = LA.norm((x_data - data),axis=1)

       # Return an array containing the indexes k closest neighbors
       neighbors_indexes = np.argsort(dist_matrix)[:k]
       return neighbors_indexes

'''
Returns single prediction based on the majority vote of neighbor
    labels
'''
def knn_clf_x(neighbors, true_labels):
       votes = []
       for idx in neighbors:
              votes.append(true_labels[idx])
       votes = np.array(votes,dtype ='int')
       return np.argmax(np.bincount(votes))
```

```
'''
K-nearest neighbors classifier implemented on test data.
'''
def knn_algorithm(train_data,train_labels,test_data,k):
        preds = []
        for x in test_data:
                neighbors = knn_x(train_data,x,k)
                preds.append(knn_clf_x(neighbors,train_labels))
        return np.array(preds)

'''
Returns unweighted errors of the classifiers
'''
def calc_error(true_labels,pred_labels):
        assert true_labels.size == pred_labels.size, "Vectors not
           equal size"
        n = true_labels.size
        miscount = 0.0
        for i in xrange(n):
                if true_labels[i] != pred_labels[i]:
                        miscount += 1.0
        return (miscount/n)

'''
Gets the training and validation error from the knn clf.
'''
def main():
        f = open('output.txt','w')
        train = np.genfromtxt('data/train_data.txt')
        test = np.genfromtxt('data/test_data.txt')
        val = np.genfromtxt('data/val_data.txt')

        # Parse data for separating training labels and dataset
        n_feat = train[0].size
        train_data = train[:,:-1]
        train_labels = train[:,n_feat-1]
        test_data = test[:,:-1]
        test_labels = test[:,n_feat-1]
        val_data = val[:,:-1]
        val_labels = val[:,n_feat-1]

        # Print training + validation error for the classifier
        k_neighbors = [1,3,5,11,16,21]
```

```
        for k in k_neighbors:
                preds_train = knn_algorithm(train_data,train_labels,
                    train_data,k)
                preds_val = knn_algorithm(train_data,train_labels,
                    val_data,k)
                preds_test = knn_algorithm(train_data,train_labels,
                    test_data,k)
                f.write("%s-neighbors: \n" % k)
                f.write("Training error: %s \n" % (calc_error(
                    train_labels,preds_train)))
                f.write("Validation error: %s \n" % (calc_error(
                    val_labels,preds_val)))
                f.write("Test error: %s \n" % (calc_error(test_labels
                    ,preds_test)))
                f.write("\n")

if __name__ == '__main__':
        main()
```

5.2 Support Vector Machine

5.2.1 Classifier

```
# Import the modules
from sklearn.externals import joblib
from sklearn import datasets
from skimage.feature import hog
from sklearn.svm import LinearSVC
import numpy as np
from collections import Counter

# Load the dataset
dataset = datasets.fetch_mldata("MNIST original")

# Extract the features and labels
features = np.array(dataset.data, 'int16')
labels = np.array(dataset.target, 'int')

# Extract the hog features
list_hog_fd = []
for feature in features:
    fd = hog(feature.reshape((28, 28)), orientations=9,
        pixels_per_cell=(14, 14), cells_per_block=(1, 1), visualise=
        False)
```

```
    list_hog_fd.append(fd)
hog_features = np.array(list_hog_fd, 'float64')

print "Count of digits in dataset", Counter(labels)

# Create an linear SVM object
clf = LinearSVC()

# Perform the training
clf.fit(hog_features, labels)

# Save the classifier
joblib.dump(clf, "digits_cls.pkl", compress=3)
```

5.2.2 Recognition

```
# Import the modules
import cv2
from sklearn.externals import joblib
from skimage.feature import hog
import numpy as np

from time import time
start_time = time()

# Load the classifier
clf = joblib.load("digits_cls.pkl")

# Read the input image
im = cv2.imread("wb.JPG")

# Convert to grayscale and apply Gaussian filtering
im_gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
im_gray = cv2.GaussianBlur(im_gray, (5, 5), 0)

# Threshold the image
ret, im_th = cv2.threshold(im_gray, 125, 255, cv2.THRESH_BINARY_INV
    )

# Find contours in the image
ctrs, hier = cv2.findContours(im_th.copy(), cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)

# Get rectangles contains each contour
```

```
rects = [cv2.boundingRect(ctr) for ctr in ctrs]

# For each rectangular region, calculate HOG features and predict
# the digit using Linear SVM.
for rect in rects:
    # Draw the rectangles
    cv2.rectangle(im, (rect[0], rect[1]), (rect[0] + rect[2], rect
        [1] + rect[3]), (0, 255, 0), 3)
    # Make the rectangular region around the digit
    leng = int(rect[3] * 1.6)
    pt1 = int(rect[1] + rect[3] // 2 - leng // 2)
    pt2 = int(rect[0] + rect[2] // 2 - leng // 2)
    roi = im_th[pt1:pt1+leng, pt2:pt2+leng]
    # Resize the image
    roi = cv2.resize(roi, (28, 28), interpolation=cv2.INTER_AREA)
    roi = cv2.dilate(roi, (3, 3))
    # Calculate the HOG features
    roi_hog_fd = hog(roi, orientations=9, pixels_per_cell=(14, 14),
        cells_per_block=(1, 1), visualise=False)
    nbr = clf.predict(np.array([roi_hog_fd], 'float64'))
    cv2.putText(im, str(int(nbr[0])), (rect[0], rect[1]),cv2.
        FONT_HERSHEY_DUPLEX, 2, (0, 255, 255), 3)

end_time = time()
time_taken = end_time - start_time
print time_taken

cv2.imshow("Resulting Image with Rectangular ROIs", im)
cv2.waitKey()
```

References

[1] Quang A. Dang, Xuan Hoai Nguyen, Hoai Bac Le, Viet Ha Nguyen and Vo Nguyen Quoc Bao. *Some Current Advanced Researches on Information and Computer Science in Vietnam*. Springer, 16-Feb-2015.

[2] Eldén, L. *Matrix Methods in Data Mining and Pattern Recognition*. Linköping University, Linköping, Sweden.

[3] Martin Law. *A Simple Introduction to Support Vector Machines*. Department of Computer Science and Engineering, Michigan State University.

[4] Wikipedia-Support Vector Machine
`https://en.wikipedia.org/wiki/Support-vector-machine`

[5] Deep Learning with H_2O on MNIST dataset (and Kaggle competition)
`http://tjo-en.hatenablog.com/entry/2015/02/25/125417`

[6] Naoki Saito. *Classifcation of Handwritten Digits*. Department of Mathematics, University of California, Davis, May 23, 2012.