OMP-BASED DATA TRAINING: A SUPERVISED MACHINE LEARNING SOLUTION FOR DICTIONARY LEARNING AND CLASSIFICATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF MATHEMATICS
OF ROSE-HULMAN INSTITUTE OF TECHNOLOGY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF SCIENCE

Steven Feng
May 2022

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Bachelor of Science.

(Dr. Kurt Bryan) Principal Adviser

Approved by the Department of Mathematics.

Preface

My intention with this thesis was to write an explanation of an intriguing subject. I chose machine learning as my subject due to its benefits to contemporary society and my passion in the field as I am doubling with a computer science major. My ultimate goal with this thesis was to possibly provide someone with a strong mathematical foundation and some expertise with MATLAB and the tools necessary to rapidly begin working on classification challenges in real life.

All the source code or files can be found on my GitHub: https://github.com/rhit-fengr/OMP-Based-Data-Training.

Acknowledgements

Thanks to my advisor, Dr. Kurt Bryan, for all of his patience and advice with me. It is his last year teaching at Rose-Hulman, but I believe no one who meets him wants to say good bye, for he will be in our hearts.

Abstract

The paper mainly discusses how a dictionary-based approach in conjunction with sparsity-based linear algebra methods can be used in classification problems in machine learning. It introduces the rationale behind the algorithm and methods to improve the dictionary with examples, including real image data of hand-written digits collected from MNIST.

Contents

List of Tables

List of Figures

Chapter 1

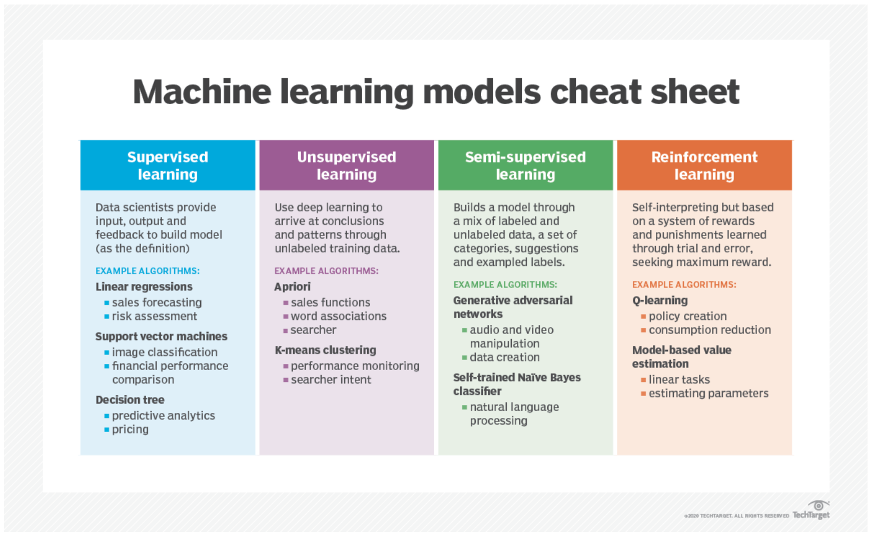
Background and Introduction

1.1 Machine Learning

Machine learning (ML) is a critical technique for achieving the objective of exploiting artificial intelligence technology, and is well-known for its learning and decision-making potential. Until the late 1970s, it was an integral aspect of the growth of artificial intelligence. It then diverged and evolved independently. Machine learning has developed into a critical response tool for cloud computing and eCommerce, and is being used to a wide variety of cutting-edge technologies.

For many firms today, machine learning is an essential part of doing business and doing research. Neural network models help computer systems improve performance through the application of algorithms. In order to make judgments without being explicitly taught, machine learning algorithms automatically develop a mathematical model using sample data – also known as "training data."

There are two big categories in machine learning: supervised and unsupervised. Supervised machine learning uses input and output with labels to accurately learn over time, while unsupervised machine learning has input and output without labels such as neural network to identify hidden patterns in data. There are also other categories such as semi-supervised learning and reinforcement learning (see Figure 1.1 for detail).



Machine learning models cheat sheet

Supervised learning

Data scientists provide input, output and feedback to build model (as the definition)

EXAMPLE ALGORITHMS:

Linear regressions
- sales forecasting
- risk assessment

Support vector machines
- image classification
- financial performance comparison

Decision tree
- predictive analytics
- pricing

Unsupervised learning

Use deep learning to arrive at conclusions and patterns through unlabeled training data.

EXAMPLE ALGORITHMS:

Apriori
- sales functions
- word associations
- searcher

K-means clustering
- performance monitoring
- searcher intent

Semi-supervised learning

Builds a model through a mix of labeled and unlabeled data, a set of categories, suggestions and exampled labels.

EXAMPLE ALGORITHMS:

Generative adversarial networks
- audio and video manipulation
- data creation

Self-trained Naïve Bayes classifier
- natural language processing

Reinforcement learning

Self-interpreting but based on a system of rewards and punishments learned through trial and error, seeking maximum reward.

EXAMPLE ALGORITHMS:

Q-learning
- policy creation
- consumption reduction

Model-based value estimation
- linear tasks
- estimating parameters

©2020 TECHTARGET. ALL RIGHTS RESERVED TechTarget

Figure 1.1: Four Categories of Machine Learning

1.2 Classification

As a branch of Machine Learning, predicting a class label from an example of input data is known as *classification*, for instance, determining if email content is spam or classifying an animal's species based on its data.

Usually, classification requires a large training dataset with numerous instances of inputs and outcomes to learn from. In most cases, it's considered as supervised machine learning since there is input data with labels for prediction use. (There are sometimes exceptions such as doing cluster analysis.)

Generally, categorized by the output, there are 4 types of classification tasks:

1. Binary Classification (KNN, Logistic Regression, SVM, Decision Trees, Naive Bayes)

2. Multi-class Classification (OCR, Face Classification)

3. Multi-label Classification (Random Forests, Multi-label Decision Trees)

4. Imbalanced Classification (Outlier Detection)

In some tasks or cases, they may be concurrent.

If categorized by the types of raw input data, there are also 4 types of classification tasks:

1. Geographical Classification
2. Chronological Classification
3. Qualitative Classification
4. Quantitative Classification

We are interested in the Optical Number Recognition problem, improving the performance of identifying a vector image of a handwritten digit, which should be categorized as a *supervised*, *multi-class* and *qualitative* classification. We should have input data with label to train on, so it is supervised; the number will vary from 0-9, so there are 10 classes in total to be *multi-class*; and since the raw input data is image format, we consider the problem itself as *multi-class* classification.

Chapter 2

The Basic Concepts

This chapter creates a framework for describing the topic and the scope of our study in order to provide a solid foundation for understanding the algorithms and dictionary method that will be discussed in the next chapters.

2.1 The USPS Digit Database

This introduces the initial plan – USPS Digit Database that we consider to train and use in a real life concern, which gives insights on the scope of our project.

Digitized pictures of several thousand handwritten digits from envelopes have been released by the United States Postal Service in the form of a database. After aligning and scaling, each sample is digitized on a 16 by 16 pixel grid. Figure 2.1 illustrates a typical example of each digit's appearance. A real number in the format of floating point is assigned to each sampled pixel in the picture. It doesn't matter how the numbers are scaled, but let's say that 0 represents black, 1 represents white, and all other values represent shades of grey. We can classify a 16 x 16 pixel picture as a vector in $\mathbb{R}^{256}$ if the 256 pixels are numbered in a specific sequence.

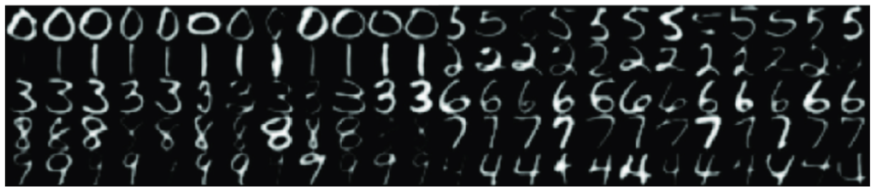


Figure 2.1: Sample digits images from the United States Postal Service (USPS) dataset.

2.2 The MNIST Digit Database

This section discusses an alternative database – MNIST Digit Database that we finally determine to train and use to solve problems in a real life concern – it can save around $600 M/year for the USPS!

MNIST (a.k.a. Modified National Institute of Standards and Technology) database, is a massive collection of handwritten digits (See figure 2.2) that is frequently used for training different image processing systems, such as facial recognition software. The database is also commonly utilized in the area of machine learning for both training and testing purposes. It was developed by "re-mixing" samples from the original datasets from the National Institute of Standards and Technology (NIST).

In total, there are 60,000 training photos and 10,000 testing images in the MNIST database. This study used half of NIST's training dataset and half of its test dataset, with the other half of NIST's testing dataset being used for the other half of the training dataset and half of the test set.

When compared to USPS data, the MNIST data source is more trustworthy and accurate for prediction when normalized to fit a 28 x 28 pixel bounding box, which is what we finally use as an alternative database. Furthermore, because the database's original creators keep a list of some of the techniques that were examined on the official article, it enables for more comparisons with our results than previously possible.

In general, several hundred examples of each of the numbers "0" up to "9" are included in the data file from which the above-mentioned sample digits are chosen.

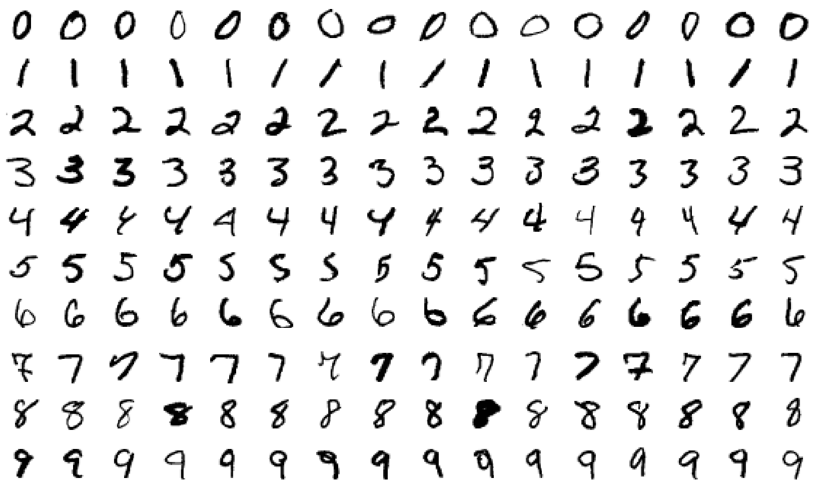


Figure 2.2: Sample digits images from the Modified National Institute of Standards and Technology (MNIST) Database.

By reading the data and training for a dictionary, an algorithm for categorizing handwritten digits automatically will be developed: when a new image for one handwritten digit is provided to the algorithm with the optimized dictionary, it identifies the number between "0" and "9".

2.3 Mathematical Formulation

To start things off, we must define some notations to simplify the concepts and terms we use and make sure we're all on the same page. We'll use **D** to denote the dictionary matrix that we generate from the original database and use for training purposes; and we'll consistently use $\mathbf{d} \in \mathbb{R}^{784}$ to represent the new digit vector that we may want to classify and recognize.

2.4 Dictionary Classification

In order to begin our classification process, we set up the equation:

$$Dx = d \tag{2.1}$$

and solve for the vector x. In our goal of using the USPS database, again, it has several hundreds of handwritten digits as unknowns to solve x, but there are 784 equations only, so the system is vastly underdetermined.

The proper way is to find the sparse solution x or even approximate sparse solution to the system. By saying *sparse solution* we mean a solution x that has very few non-zero components, so that the scope of problem can be narrowed down. The component x_i in the solution to (2.1) tells us the weight of the ith item (the ith column representing a sample digit) in D used to manufacture d as a linear combination, which makes logical sense. If we discover that the majority of the solution x vector's nonzero x_i components correspond to items of a certain class, such as handwritten "7"s, we would assume that d also corresponds to a "7," since **d** fits the synthesis from 7's very well.

However, perhaps expressing a "7" as a linear combination of other types takes more samples and hence a less sparse solution, and we will introduce certain technique and expand on the process of precisely determining the digit later.

2.5 A Small Example for Illustration

Suppose we have a very simple dictionary D with two handwritten digits: 0 and 1 needed for distinction. Suppose we have one instance (a 2×1 vector image – assume v_i for digit i) for each type of digit, where:

$$v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, v_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} . \tag{2.2}$$

Then

$$D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} . \tag{2.3}$$

Now we scan a handwritten digit (0 or 1) and get the data vector $d = \begin{bmatrix} 2.20 \\ 0.07 \end{bmatrix}$ and want to use the dictionary D to identify it. We will set up the equation:

$$Dx = d \Leftrightarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x = \begin{bmatrix} 2.20 \\ 0.07 \end{bmatrix} \tag{2.4}$$

.

Solve it, and the solution is $x = \begin{bmatrix} 2.20 \\ 0.07 \end{bmatrix}$. In this very simple case, an intuitive solution is to choose the most similar pattern – since 2.2 is significantly greater, we can ignore 0.07 and say it is classified as the class of digit 0. It is exactly the same process of finding the term with greatest weight in the linear combination of x composed by v_i for any possible digit i.

Chapter 3

Finding Sparse Solutions for Linear Systems

For finding sparse solutions for the system (2.1), we have some basic classes of optimization algorithm that are effective.

3.1 Basis Pursuit

Basis pursuit (BP), as a basic problem of linear optimization, address the problem of the form:

$$\begin{array}{l} \min \Sigma_{i=1}^{n}|x_i| \\ \text{subject to } Ax = b \end{array} . \tag{3.1}$$

A is $M \times N$ transform matrix. **b** is $M \times 1$ vector of observation, and it wants to minimize the l^1 norm of **x** subject to linear constraints $Ax = b$. Basis pursuit is a common method of seeking sparse solutions to a linear system $Ax = b$. In this text we will not use basis pursuit, but an alternative approach called *Orthogonal Matching Pursuit.*

3.2 Orthogonal Matching Pursuit

Realizing that there is usually noise in the data vector b and BP algorithm ends up doing over-fitting, Orthogonal Matching Pursuit (OMP), is an alternative to BP that uses a greedy approach to seek a sparse solution to $Ax = b$. Consider the problem of minimizing the quantity

$$\min ||Ax - b||_2 \ . \tag{3.2}$$

Again, A is a $M \times N$ transform matrix, and b is a $M \times 1$ vector of observations. We seek that vector x with sparsity bounded by some positive integer m (that is, x has at most m nonzero components) that minimizes $||Ax - b||_2$.

3.3 OMP Algorithm

For $Ax = b$, Orthogonal Matching Pursuit (OMP) uses a *greedy algorithm* that accumulates sparse solutions. It initiates: $x_0 = 0$, residual $r_0 = 0$ and index support set $S_0 = \varnothing$. We iteratively add indices to S_0 by finding

$$i_{k+1} = \arg\max_{1 \leq j \leq n} |r^k \cdot A_j| \ . \tag{3.3}$$

and update S_{k+1} by

$$S_{k+1} = S_k \cup i_{k+1} \ . \tag{3.4}$$

Note that A_j is the jth column of A. Correspondingly, x_{k+1} will also be updated by

$$x_{k+1} = \arg\min_{x \in \mathbb{R}^{256}, supp(x) \subseteq S_{k+1}} ||b - Ax_k||_2 \tag{3.5}$$

The iteration stops when the norm of the residual $r_{k+1} = b - Ax_{k+1}$ is smaller than a certain tolerance or upper bound of the sparsity for the solution x_k is reached. There are concerns with finding sub-optimal solutions when using OMP, since the index i_k added to support set S_k cannot be removed even if it's overdoing (the OMP itself doesn't trace back even if it trains a dictionary worse than the previous iteration). For this reason, there are modified OMP such as "Band-excluded Locally Optimized Orthogonal Matching Pursuit"(BLO-OMP). However, the drawback of OMP can be fixed by iterations and two techniques we will introduce later in the chapter. And considering the cheapness the efficiency, OMP suffices to solve our problem.

Chapter 4

Dictionary Learning

Suppose we have a raw dictionary D generated from the database that is a $d \times N$ matrix. Since the dataset needs to have multiple samples for each class, it's reasonable to assume that $N >> d$. So, D contains N columns of d-dimensional vectors. Remember we have the system $Dx = d$ (2.1) to solve.

4.1 Classification Approach

4.1.1 Introduction

Knowing the vector solution $x \in \mathbb{R}^N$, the approach starts by introducing a new operator $\delta_j(x)$ that maps x to another vector in $\mathbb{R}^N$ by zeroing out all components of x except for those that correspond to class j vectors in the dictionary D:

$$\delta_j(x) =< 0, 0, ..., x_{N_{j-1}+1}, ..., x_{N_j}, 0, ..., 0 > \ . \tag{4.1}$$

where $1 \leq j \leq C$ (C represents the number of classes in the dictionary). Notice that when all non-zero components of x belong to class k vectors, we have:

$$\delta_k(x) = \begin{cases} x, & \text{when } k = j \\ 0, & \text{when } k \neq j \end{cases} \quad .$$

To make it simple to illustrate, we define a function $S_j = ||D\delta_j(x) - d||_2^2$ based on this for $1 \leq j \leq C$. So if data d falls in class j, we should have $Dx \approx d \Rightarrow D\delta_j(x) \approx d \Rightarrow S_j \approx 0$. Thus, we can classify the data vector d as in class k if

$$k = \arg\min_{1 \leq j \leq C} S_j(x). \tag{4.2}$$

4.1.2 Example Code and Results

In "dict_race_new.m" code, we have 4 types of functions (i.e. 4 classes). With a noise level of 0.05, around 85% of data can be classified correctly. With a noise level of 0.2, over 50% of data can be classified generally. This is good enough for most data with a noise level less than 0.05.

4.2 Gradient Approach

4.2.1 Introduction

It can be beneficial to iteratively modify the dictionary D, to improve the performance of this classification scheme. To keep optimizing the dictionary, we introduce a quadratic function first:

$$E(D) = \tfrac{1}{2}||DA - X||_{fro}^2. \tag{4.3}$$

where the dictionary D is $d \times N$, the matrix A is $N \times M$, and the training data X is $d \times M$. $E(D)$ serves here as reflection on how well the dictionary performs on the training data, because it is intuitive that the $||DA - X||$ is no more than organizing the equation $||DA = X||$ and taking one side of it to see how much it is off from 0. And we choose Frobenius norm (a.k.a Euclidean norm for the matrix).

Similar to function S_j in the previous approach, this function serves as an evaluation of the dictionary D. Since we want to optimize and find the minimum of it, we are interested in the derivative $\partial E/\partial D_{pq}$ for fixed indices p and q in the dictionary D where $1 \le p \le d$ and $1 \le q \le N$.

Given the context of our problem, we have the function $E(D)$ as:

$$E(D) = \textstyle\sum_{i=1}^{M} E_i(D). \tag{4.4}$$

where

$$E_i(D) = \tfrac{1}{2}||Dx_i - d_i||_{fro}^2. \tag{4.5}$$

Note that x_i is the ith column of A and d_i is the ith column of training data matrix X.

By definition of matrix-vector multiplication, we can rewrite the equation 4.5 as:

$$E_i(D) = \tfrac{1}{2}\textstyle\sum_{j=1}^{d}(\sum_{k=1}^{N} D_{jk}x_k - d_j)^2 \ . \tag{4.6}$$

Computing the derivative of (4.6) with respect to D_{pq}, we have:

$$\tfrac{\partial E_i}{\partial D_{pq}} = x_q(\textstyle\sum_{m=1}^{N} D_{pm}x_m - d_p) \ . \tag{4.7}$$

Regarding (4.7) in the form of $\frac{\partial E_i}{\partial D_{pq}} = x_q v_p$ where $v = Dx - d$, we can safely rewrite it as:

$$[\tfrac{\partial E_i}{\partial D_{pq}}] = (Dx - d)x^T \ . \tag{4.8}$$

4.2.2 Minimization Process

By summing up E_i and setting $\frac{\partial E}{\partial D} = 0$ to solve, we get the equation:

$$Dxx^T = dx^T \ . \tag{4.9}$$

Defining $Q = \sum_{i=1}^{M} x_i x_i^T$ and $R = \sum_{i=1}^{M} d_i x_i^T$, from (4.9) we have:

$$DQ = R \ . \tag{4.10}$$

Q will be $N \times N$ and each x_i is sparse. R will be $d \times N$. Solving by $D = RQ^{-1}$ gives the minimizer for $E(D)$.

It is worthwhile to note that since Q is positive definite symmetric, it is preferred to use Cholesky Decomposition to solve the linear system and get the minimizer.

4.2.3 MATLAB Implementation

In "optD.m"code (which is actually 3 lines), we address the approach and let Matlab handle the process of solving (4.10). A simple and integral example comes up in the next chapter.

Chapter 5

General Dictionary Procedure

5.1 Introduction with Outline

With both approaches used as introduced in the last chapter, the outline below should suffice to illustrate how they integrate together and what is going on:

1. Use a basic dictionary D generated from anywhere and a training data matrix d to get started. Both should have attached class labels.
2. Find the sparse solution for each d_i to Dx_i, and construct x as the matrix with $\delta_c i(x_i)$ for each ith column.
3. Minimize $||Dx - d||^2_{fro}$ with respect to D and get the dictionary improved once.
4. Repeat step 2-3 until any iteration limit is reached or the process generates insufficient progress.

5.2 Mock Example and Results

The script "make_data.m" generates a 10×20 initial dictionary and a 10×200 training data matrix with random seed set to 3050, using sum of Gaussians as the signal input, per step 1 in the outline.

The script "classifyN.m" loads all training data, the dictionary, and keeps doing work from 2-4 until improvement is smaller than the tolerance (we use the error difference ratio to evaluate this if it should break).

In the relatively small example, the performance of the dictionary got improved by

15.42% in 5 iterations only, which shows the significant improvement.

In the next stage, we are going to import real MNIST handwritten digits database to test. The modified OMP algorithm and further investigations into step 3 in the outline provides us insights for a better performance.

Chapter 6

Test With MNIST Data

This chapter reports how we processed MNIST Data, the scope of our training process, how we test, and the result compared to other method for MNIST digit classification.

6.1 Pre-processing with MNIST Data

The original data can be downloaded from the MNIST official website: `http://yann.lecun.com/exdb/mnist`. However, those .gz files need to be transformed first to better adapt the readable files that can be accepted by MATLAB.

See my Google Colab work (click to see it). It transforms those compressed data into raw files.

We implement MNISTData.m to call $mnist\_parse(...)$ function and parse raw files, normalize all data, and finally save them as formatted .mat files that are friendly to load.

6.2 Training Process

In classify_digits.m, we will load the trainingdata.mat which has the size of 784×60000 large data. We select 2000 of them for training and a dictionary size of 100 (or, 784 × 100) to apply training process described in Chapter 5. We will also load the testing data of size 784 × 10000 for validation test use.

As a very satisfactory result, it runs 156.027 seconds for loading all data and running 10 iterations of improving the dictionary quality by having training score of 88.63% to 96.21%, and having validation score of 88.65% to 94.92%, which is around 7% improvement in the dictionary accuracy – not even counting the first time when generating a good dictionary. The outputs can be found in the index to look at in detail with code.

6.3 Results and Observations

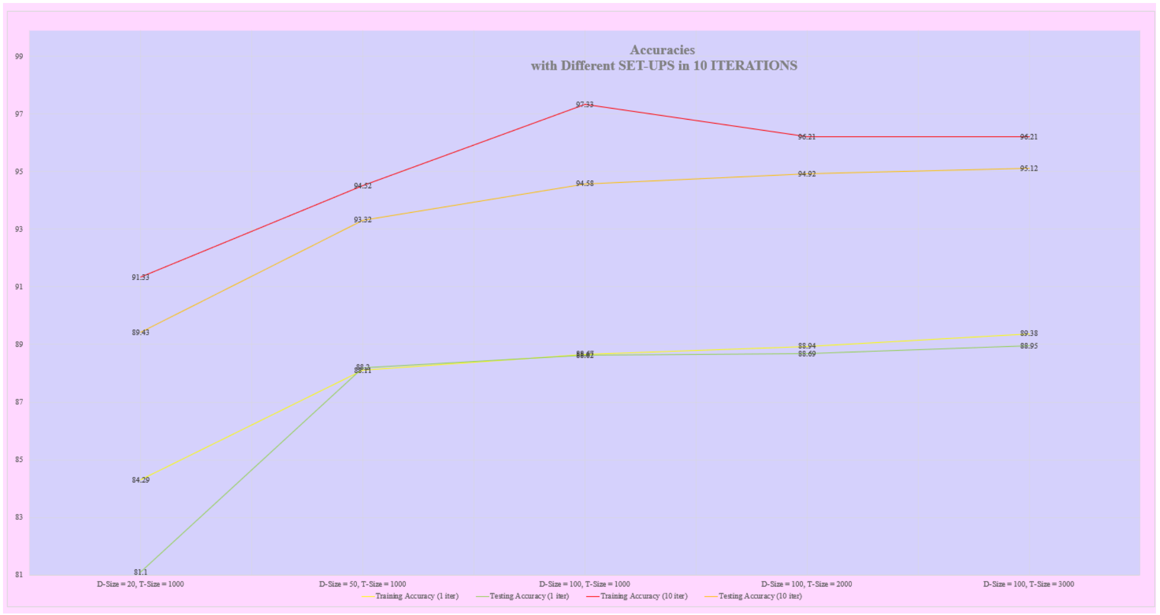


Figure 6.1: MNIST Training Results With Different Setups In 10 Iterations

In our five experiments of ten iterations, we altered or increased dictionary size and training and testing size – the first three stages increase dictionary size from 20 to 100, while the last three stages increase training size and testing size from 100 to 3000.

The following observations can be noted:

1. Greater dictionary size generally produces higher accuracy.
2. Greater training data size generally produces higher accuracy.
3. Dictionary size has larger impact on the accuracy compared to training data size - a reasonable size of dictionary used in training period can makes the testing accuracy stable, even if the train and testing data size increases.
4. The training accuracy decreases in the last three stages because it goes harder for the same size of dictionary to fit, but a reasonable size of dictionary still survive since the testing accuracy is stable and does not even decrease (rationale as illustrated in 3).

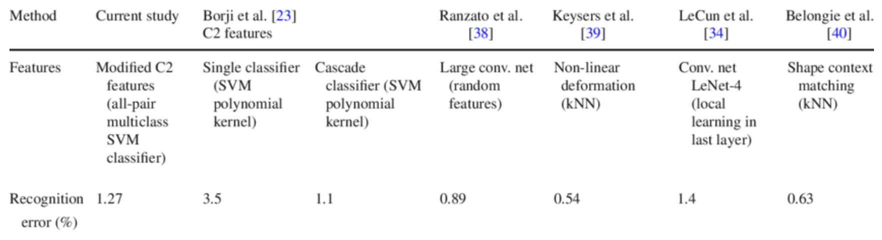
The success rate looks very good overall in small number of iterations, and also small time.

A table is attached here for the specific summary of our method performance after 10 iterations:

Dictionary Size	Data Size	Training Acc (%)	Testing Acc(%)	Time (s)
20	1000	91.33	89.43	60.79
50	1000	94.52	93.32	90.51
100	1000	97.33	94.58	117.15
100	2000	96.21	94.92	126.09
100	3000	96.21	95.12	138.64

6.4 Comparison & Conclusion

As the figure above indicated, this OMP method alone does not show competency in efficiency or recognition rate. However, it does have hardware barrier or device limitation for model training such as KNN and neural networking, and will provide convenient and responsive service to users in real life, apart from getting connected to the server (such as a vending machine). And another reason could be that I tested



Method	Current study	Borji et al. [23] C2 features		Ranzato et al. [38]	Keysers et al. [39]	LeCun et al. [34]	Belongie et al. [40]
Features	Modified C2 features (all-pair multiclass SVM classifier)	Single classifier (SVM polynomial kernel)	Cascade classifier (SVM polynomial kernel)	Large conv. net (random features)	Non-linear deformation (kNN)	Conv. net LeNet-4 (local learning in last layer)	Shape context matching (kNN)
Recognition error (%)	1.27	3.5	1.1	0.89	0.54	1.4	0.63

Figure 6.2: Comparison on different methods' error rate for handwritten digit recognition over MNIST dataset.

with 10 iterations beginning with sample size of 100 only since I am using a computer, which significantly reduce the success rate of our method also.

Overall, the result is satisfactory with my small machine that runs the OMP algorithm.

Bibliography

[1] Kurt Bryan, verbal and email communications.

[2] J. Brownlee, *4 Types of Classification Tasks in Machine Learning*, Retrieved from: https://machinelearningmastery.com/types-of-classification-in-machine-learning.

[3] Kassidy Kelley, *Top 9 types of machine learning algorithms, with cheat sheet*, Retrieved from: https://www.techtarget.com/searchenterpriseai/feature/5-types-of-machine-learning-algorithms-you-should-know.

[4] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar, *Introduction to Data Mining*, Pearson Education India, 2016.

[5] Hamidi, Mandana & Borji, Ali. *Invariance analysis of modified C2 features: Case study-handwritten digit recognition.* Mach. Vis. Appl.. 21. 969-979. 10.1007/s00138-009-0216-9.

[6] Wright, John, Allen Y. Yang, Arvind Ganesh, S. Shankar Sastry, and Yi Ma. *Robust face recognition via sparse representation.* IEEE transactions on pattern analysis and machine intelligence 31, no. 2 (2008): 210-227.

[7] Alcin, Omer F., Abdulkadir Sengur, Jiang Qian, and Melih C. Ince. *OMP-ELM: orthogonal matching pursuit-based extreme learning machine for regression.* Journal of Intelligent Systems 24, no. 1 (2015): 135-143.

[8] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits.* Retrieved from: http://yann.lecun.com/exdb/mnist/.

Appendix A

Source Code (MATLAB)

classify_digits.m: the main algorithm that runs the training methods and report the results .

```
%Code to perform classification on handwritten digits data.
clear;
clc;

tic;

%Maximum number of iterations to improve dictionary
maxits = 10;

%Regularization parameter
%reg = 0.0;

%Sparsity limit for OMP
sparsity = 5;

%OMP parameters
opts.omp = 1; %Use OMP
opts.maxits = sparsity; %Sparsity limit
opts.verbose = 0;

%Load in training data
load('trainingdata.mat');

%Normalize all columns, any other preprocessing.
[n1,n2] = size(X);
for k=1:n2
    X(:,k) = X(:,k)/norm(X(:,k),2);
end
```

```
%Load in testing data
load('testingdata.mat');

%Normalization for testing data
[t1,t2] = size(T);
for k=1:t2
    T(:,k) = T(:,k)/norm(T(:,k),2);
end

%Partition the training data into a dictionary D
%and proper training data X.
D1 = 100; N1 = 2000;
D = X(:,1:D1); %About 10 percent of each digit appears.
%It will be convenient to use the label "1" for the digit "0"
%and more generally label "i" for digit "i-1".
Datom_labels = 1+label1(1:D1);


%Relegate the remaining training data to be actual training data.
X = X(:,D1+1:N1);
cat_labels = 1+label1(D1+1:N1);

Tcat_labels = 1+label2(:,:);

clear label1 label2;

%Dictionary, training data matrix dimensions
[d,M] = size(X);
[~,N] = size(D);
[~,MT] = size(T);

%For later convenience, sift out indices for each class, use to
%construct delta operators. Note that F{i} is for the digit i-1.
C = max(cat_labels); %Number of classes
F = cell(1,C);

for i=1:C
    F{i} = (Datom_labels.*(Datom_labels==i)/i)'; %These should be column vectors
end

%This function will restrict support of a vector to appropriate
%class label indices.
delta = cell(1,C);
for i=1:C
    delta{i} = @(v) v.*F{i};
end

%Iterate to improve dictionary. The matrix A is as described
```

```
%previously in a LaTeX document, the solutions to D*A = X.
A = zeros(N,M);
AT = zeros(N,MT);

for jj=1:maxits
  %Run OMP on each sample. Install results into matrix "A".

  for i=1:M
    %Call OMP to solve D*alpha = x
    [alpha, rhist] = omp(D, X(:,i), opts);

    %Restrict support of alpha
    A(:,i) = delta{cat_labels(i)}(alpha);
  end

  for i=1:MT
    %Call OMP to solve D*alpha = x
    [alpha, rhist] = omp(D, T(:,i), opts);

    %Restrict support of alpha
    AT(:,i) = delta{Tcat_labels(i)}(alpha);
  end

  %Routine to compute percentage correctly classified.
  fprintf("[Pre-iteration %d] %f percent of training data is classified correctly\n",
      jj,correct_class(D,A,X,cat_labels,Datom_labels));
  fprintf("                   %f percent of testing data is classified correctly\n",
      correct_class(D,AT,T,Tcat_labels,Datom_labels));




  %How good is this dictionary?
  ED = norm(D*A-X,'fro')/norm(X,'fro');
  fprintf("                   Dictionary error metric is %f\n", ED);

  %Now let's optimize the dictionary
  Dopt = optD(A, X);

  ED2 = norm(Dopt*A-X,'fro')/norm(X,'fro');
  fprintf("                  Improved dictionary error metric is %f\n", ED2);

  %Update dictionary
  D = Dopt;

  %Renormalize columns
  for k=1:N
      D(:,k) = D(:,k)/norm(D(:,k),2);
  end


  %Compute percent correctly classified, print
  fprintf("[Post-iteration %d] %f percent of training data is classified correctly\n", ...
      jj,correct_class(D,A,X,cat_labels,Datom_labels));
  fprintf("                  %f percent of testing data is classified correctly\n", ...
      correct_class(D,AT,T,Tcat_labels,Datom_labels));

end

toc;
```

classify_class.m: the algorithm of checking success rate for classification .

```
%Compute what percentange of training sample classified correctly.
function rate = correct_class(D,A,X,cat_labels,Datom_labels)

%Dictionary, training data matrix dimensions
[~,M] = size(X);
%[~,N] = size(D);

%For later convenience, sift out indices for each class, use to
%construct delta operators.
C = max(cat_labels); %Number of classes
F = cell(1,C);
for i=1:C
    F{i} = (Datom_labels.*(Datom_labels==i)/i)'; %These should be column vectors
end

%This function will restrict support of a vector to appropriate
%class label indices.
delta = cell(1,C);
for i=1:C
    delta{i} = @(v) v.*F{i};
end

%In the end, what fraction of training sample correctly
%classified?
classif = zeros(1,M);
thing = zeros(1,C); %Working space
succ = 0;
for i=1:M %Loop over samples
    for j=1:C
        alpha = delta{j}(A(:,i));
        thing(j) = norm(D*alpha-X(:,i),2);
    end
    [~,urp] = min(thing);
    classif(i) = urp;
    if classif(i) == cat_labels(i)
        succ = succ+1;
    end
end

% fprintf("%f classified correctly\n", 100*succ/M);
rate = 100*succ/M;
```

omp.m: the orthogonal matching pursuit algorithm .

```
function [xsol, rhist] = omp(A, b, opts)
%Usage: [xsol, rhist] = omp(A, b, opts)
%Given matrix A and vector b, run omp (or just mp) to solve A*x = b.
%Returns solution in "xsol", residual history in "rhist".

%Function to read options
function out = setOpts( field, default )
    if ~isfield( opts, field )
        opts.(field)    = default;
    end
    out = opts.(field);
end

%Scale the columns of A; compensate later.
[m,n] = size(A); %A is m by n
A2 = zeros(size(A));
s = zeros(n,1);
for i=1:n
    s(i) = norm(A(:,i),2);
    A2(:,i) = A(:,i)/s(i);
end

%Set options
do_omp = setOpts('omp', 1); %Default to OMP
verbose = setOpts('verbose', 0);
maxits = setOpts('maxits', floor(m/2));
coher = setOpts('coher', 0);
tol = setOpts('tol', 1.0e-10);

%If requested, computed and print coherence
if coher
    thing = abs(A2'*A2-eye(n));
    coh = max(max(thing));
    fprintf('Coherence is %f\n', coh);
end
```

```
k = 0; %Loop counter
xsol = zeros(n,1); %Prospective solution
S = []; %Support indices of tentative solution

%Loop, build up solution one index a time
while k<=maxits
    if verbose
        fprintf('Estimated Solution '); disp(xsol');
        pause
    end
    resid = b - A2*xsol; %Residual vector
    rhist(k+1) = norm(resid,2); %Record its norm
    if verbose
        fprintf('Residual Vector '); disp(resid');
        pause
    end
    %fprintf('Iteration %d residual %f\n',k,rhist(k+1));
    if rhist(k+1) < tol || k==maxits %Quit if small enough
        break;
    end
    c = A2'*resid;
    if verbose
        fprintf('c = AT*r  '); disp(c');
        pause;
    end

    [ci,ii] = max(abs(c));
    if verbose
        fprintf('max ci = %f Adding index %d to support\n', ci, ii);
        pause
    end

    %Matching Pursuit
    xsol(ii) = xsol(ii) + c(ii);
    S = union(S,ii);

    if verbose
        fprintf('Estimated support S = '); disp(S);
        pause
    end

    %Or do OMP if flag set
    if do_omp == 1
        %Implement the least squares solution on suppport set S.
        A3 = A2(:,S);
        xsol3 = A3\b;
        xsol(S) = xsol3;
    end

    k = k+1;
end

%Adjust xsol back due to scaling
xsol = xsol./s;

end
```

optD.m: the algorithm that help optimize the dictionary we trained.

```
function optD = optD(A, X)

% delta = 5;

%v1
optD = X/A;

% %v2
% Q = A*A';
%
% R = X*A';
%
% optD = R/Q;

%v3
% Q = A*A';
% Q = Q+delta*eye(size(Q));
%
% R = X*A';
%
% %May want pseudoinverse?
% optD = R/Q;
```

MNISTData.m: the algorithm that parse and generate training and testing dataset from MNIST image data file.

```
[trainingdata, traingnd] = mnist_parse('train-images-idx3-ubyte', 'train-labels-idx1
trainingdata = double(reshape(trainingdata, size(trainingdata,1)*size(trainingdata,2
X=trainingdata';
traingnd = double(traingnd);
label1=traingnd';
M = size(X,2);
% %Normalize training data to unit length
% for k=1:M
%     X(:,k) = X(:,k)/norm(X(:,k),2);
% end
% save('trainingdata.mat','X','label1');

X2=X(:,1:M/10);
label11=label1(:,1:M/10);
save('small-trainingdata.mat','X2','label11');


[testdata, testgnd] = mnist_parse('t10k-images-idx3-ubyte', 't10k-labels-idx1-ubyte');
testdata = double(reshape(testdata, size(testdata,1)*size(testdata,2), []).');
T=testdata';
testgnd = double(testgnd);
label2=testgnd';
save('testingdata.mat','T','label2')

d = 784;
N = 100;

%Dictionary, training data matrix dimensions
[d,M] = size(X);

D = zeros(d,N);
Dlabel = zeros(1,N);

% 0
ccc = 1;
for j=0:9
    for k=1:M
        if(label1(k)==j)
            D(:,ccc) = X(:,k);
            Dlabel(ccc)=j;
            ccc=ccc+1;
        end
        if(ccc>(N-1)*(j+1)/10+1) break;
        end
    end
end

% %Normalize atoms to unit length
% for k=1:N
%     D(:,k) = D(:,k)/norm(D(:,k),2);
% end

save('mid-dictionary.mat','D','Dlabel')
```

mnist_parse.m: the helper method for parsing MNIST data into matrices.

```
function [images, labels] = mnist_parse(path_to_digits, path_to_labels)

% Open files
fid1 = fopen(path_to_digits, 'r');

% The labels file
fid2 = fopen(path_to_labels, 'r');

% Read in magic numbers for both files
A = fread(fid1, 1, 'uint32');
magicNumber1 = swapbytes(uint32(A)); % Should be 2051
fprintf('Magic Number - Images: %d\n', magicNumber1);

A = fread(fid2, 1, 'uint32');
magicNumber2 = swapbytes(uint32(A)); % Should be 2049
fprintf('Magic Number - Labels: %d\n', magicNumber2);

% Read in total number of images
% Ensure that this number matches with the labels file
A = fread(fid1, 1, 'uint32');
totalImages = swapbytes(uint32(A));
A = fread(fid2, 1, 'uint32');
if totalImages ~= swapbytes(uint32(A))
    error('Total number of images read from images and labels files are not the same
end
fprintf('Total number of images: %d\n', totalImages);

% Read in number of rows
A = fread(fid1, 1, 'uint32');
numRows = swapbytes(uint32(A));

% Read in number of columns
A = fread(fid1, 1, 'uint32');
numCols = swapbytes(uint32(A));

fprintf('Dimensions of each digit: %d x %d\n', numRows, numCols);

% For each image, store into an individual slice
images = zeros(numRows, numCols, totalImages, 'uint8');
for k = 1 : totalImages
    % Read in numRows*numCols pixels at a time
    A = fread(fid1, numRows*numCols, 'uint8');

    % Reshape so that it becomes a matrix
    % We are actually reading this in column major format
    % so we need to transpose this at the end
    images(:,:,k) = reshape(uint8(A), numCols, numRows).';
end
```

```
% Read in the labels
labels = fread(fid2, totalImages, 'uint8');

% Close the files
fclose(fid1);
fclose(fid2);

end
```