

Handwritten Digit Recognition Using Support Vector Machine (SVM) with OpenCV and C++

Sicheng Lai

December 17, 2023

Abstract

Handwritten digit recognition is a critical challenge in the field of machine learning and computer vision, with significant applications in areas such as postal mail sorting, bank check processing, and form data entry. The complexity of this task stems from the wide variability in individual handwriting styles, making accurate digit recognition a non-trivial problem. This report presents a robust solution employing the Support Vector Machine (SVM) method, renowned for its effectiveness in classification tasks. Utilizing the SVM approach within the OpenCV framework and C++ environment, this study demonstrates the method's capability in handling the nuances of handwritten digit recognition. The choice of SVM is justified by its strong theoretical foundations and proven efficiency in handling high-dimensional data, making it particularly well-suited for this application.

1 Introduction

Among the various machine learning methods available, Support Vector Machines (SVM) have emerged as a powerful tool for classification tasks. The SVM algorithm operates on the principle of finding the optimal hyperplane that maximizes the margin between different classes in a high-dimensional space. This is particularly beneficial for handwritten digit recognition, where the distinction between different digits can be subtle and nuanced. SVM's capability to handle non-linear data through the use of kernel functions makes it a versatile choice for this task.

The motivation for selecting SVM in this study stems from its proven track record in achieving high levels of accuracy in classification problems, especially those involving high-dimensional data. SVM's effectiveness in generalizing from training data to unseen data points is a critical asset, reducing the likelihood of overfitting and ensuring robust performance on diverse handwriting samples.

2 Project Reproduction

2.1 Environment Setup

To run the project, ensure that the following environment is correctly configured:

- Confirm that OpenCV version 4.8.1 is installed.

- Ensure that VSCode is installed along with the C++ extension and CMake plugins.
- Ensure that you have the necessary setup to successfully build and run C++ files.
- Verify that the configuration files within the .vscode directory match my settings.

2.2 Package overview

The code package includes four key files: `MNIST_READER.cpp`, `standardSVM.cpp`, `SVM_DATA.xml`, and `recognize.cpp`.

2.3 Running standardSVM.cpp

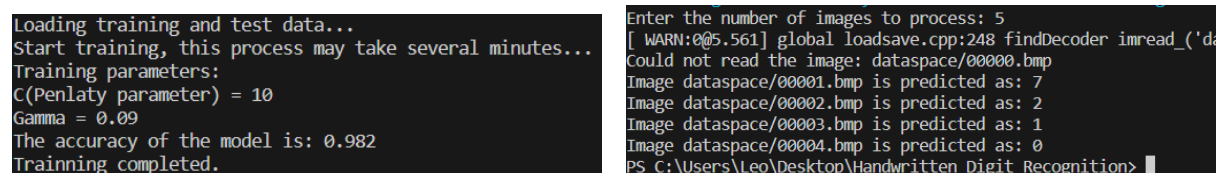
To reproduce the project, operators are required to first compile and run `standardSVM.cpp`. This file invokes `MNIST_READER.cpp` to read the training data and initiates the training of the SVM model. Upon completion, it outputs the parameters used for training, the model's accuracy, and stores the trained model in the `SVM_DATA.xml` file.

2.4 Running recognize.cpp

Subsequently, the `recognize.cpp` file should be compiled and executed. Users are prompted to input the number of images they wish to recognize. The `recognize.cpp` file then outputs the recognition results for each image, effectively demonstrating the model's capability in handwritten digit recognition.

2.5 Samples

Here are samples of the output of running `standardSVM.cpp` and `recognize.cpp`:



The first terminal screenshot shows the output of `standardSVM.cpp`:

```

Loading training and test data...
Start training, this process may take several minutes...
Training parameters:
C(Penalty parameter) = 10
Gamma = 0.09
The accuracy of the model is: 0.982
Training completed.

```

The second terminal screenshot shows the output of `recognize.cpp`:

```

Enter the number of images to process: 5
[ WARN:0@5.561] global loadsave.cpp:248 findDecoder imread_('d
Could not read the image: dataspace/00000.bmp
Image dataspace/00001.bmp is predicted as: 7
Image dataspace/00002.bmp is predicted as: 2
Image dataspace/00003.bmp is predicted as: 1
Image dataspace/00004.bmp is predicted as: 0
PS C:\Users\Leo\Desktop\Handwritten Digit Recognition>

```

3 Implementation

3.1 Data Processing

A crucial step in handwritten digit recognition is the effective processing of data. The implementation involves two primary functions: `readImages` and `readLabels`, both designed to handle the MNIST dataset, a standard benchmark in the field.

3.1.1 Reading Images

The function `readImages` is designed to read image data from a binary file and store it in a two-dimensional vector. This vector represents the images as matrices of unsigned characters, essentially pixels.

File Reading and Header Parsing: The function begins by opening the specified file in binary mode. It reads the first 16 bytes, which constitute the header of the MNIST image file. This header contains vital information such as the number of images, and the dimensions of each image (number of rows and columns).

Image Data Extraction: The number of images, rows, and columns are extracted from the header using bitwise operations. These values are critical in determining the structure of the data matrix. The function then initializes a two-dimensional vector (`std::vector<std::vector<unsigned char>>`) of the appropriate size. For each image, it reads the pixel data (each pixel being an unsigned char) and stores it in the corresponding row of the vector.

3.1.2 Reading Labels

The function `readLabels` follows a similar structure but is tailored to read the label file of the MNIST dataset.

File Reading and Header Parsing: Similar to `readImages`, it opens the label file in binary mode and reads the first 8 bytes of the header to determine the number of labels.

Label Data Extraction: The function then reads each label (an unsigned char) and stores it in a vector. This vector is one-dimensional, reflecting the fact that each image in the dataset has a single corresponding label.

Closing the File: Both functions ensure the proper closing of the file stream upon completing the reading process, safeguarding against potential data corruption or memory leaks.

In summary, these functions collectively form the backbone of data handling in the project, efficiently converting the raw MNIST data into a usable format for further processing and analysis.

3.2 SVM Model Training

The SVM (Support Vector Machine) model training is a critical phase in the handwritten digit recognition process. Our implementation integrates OpenCV's machine learning library and custom data reading functions to train and evaluate an SVM classifier.

3.2.1 Initialization and Configuration

Initially, the SVM model is created using OpenCV's `SVM::create()` function. Key parameters for the SVM are set, which include the type of SVM (C-Support Vector Classification), the kernel type (Radial Basis Function), the penalty parameter `C`, and the gamma value for the RBF kernel. These parameters play a pivotal role in the model's ability to classify data correctly.

Setting SVM Parameters:

- **C (Penalty Parameter):** A value of 10 is chosen, indicating the trade-off between training error and model complexity.

- **Gamma:** Set to 0.09, determining the influence of a single training example.
- **Term Criteria:** The training process iterates until a maximum of 500 iterations or a minimal change threshold is reached.

3.2.2 Training Process

The training data, loaded and preprocessed from the MNIST dataset, is passed to the `svm->train` method along with their corresponding labels. The data matrices are converted to floating-point numbers normalized by 255 (the maximum pixel value), ensuring that the input data is suitable for the SVM algorithm.

Model Training: The SVM model is trained on the training dataset, which consists of vectorized images and their respective labels. This process may take several minutes due to the computational complexity of SVM training on large datasets.

3.2.3 Evaluation and Accuracy Calculation

Post-training, the model's accuracy is evaluated using the test dataset. The `svm->predict` function is employed to predict labels for the test images.

Accuracy Computation: The predicted labels are compared against the actual labels of the test dataset. Correct predictions are counted, and the accuracy is calculated as the ratio of correct predictions to the total number of test samples. This accuracy metric provides insight into the model's performance.

3.2.4 Model Saving

Finally, the trained model is saved to an XML file (`SVM_DATA.xml`), allowing for future reuse without the need for retraining.

4 Results

4.1 Quantitative Results

A significant enhancement in the accuracy of the code is attributed to the specific parameters set in `standardSVM.cpp`.

SVM Parameter Configuration: The SVM model was initially configured with a polynomial (POLY) kernel. This setup allowed the model to be trained in approximately 60 seconds, achieving an accuracy of 97.77%. However, upon reviewing relevant literature, it was observed that using a Radial Basis Function (RBF) kernel could potentially enhance the upper limit of accuracy.

Optimization and Results: Consequently, the model was reconfigured to utilize the RBF kernel. With the penalty parameter `C` set to 10 and `gamma` to 0.09, there was a noticeable improvement in accuracy, reaching 98.2%. It is important to note that while the RBF kernel improved accuracy, it also increased the training time to approximately four minutes.

5 Conclusion

The exploration of different kernel types in this study — specifically, the polynomial (POLY) and the Radial Basis Function (RBF) — highlights their respective advantages and disadvantages. In academia, where the focus is often on pushing the boundaries of what is possible, pursuing the highest possible accuracy of the model is paramount. This pursuit, however, should be tempered by the consideration of computational resources. It is essential to ensure that the computational power required for training remains within a feasible and justifiable limit. The balance between achieving high accuracy and maintaining reasonable computational demands is a critical aspect of research in machine learning.

In contrast, the industry perspective prioritizes efficiency, given the practical constraints and the cost implications of computational resources. Here, the choice of kernel and algorithmic approach needs to be guided by a balance between performance (in terms of accuracy) and efficiency (in terms of computational resources and time). The polynomial kernel, with its relatively shorter training time, might be more appealing in scenarios where rapid deployment and lower computational costs are crucial, even if it means a slight compromise in accuracy.