

Capstone Project

Brice Nkengsa

Machine Learning Engineer Nanodegree

October 30th, 2016

1. Definition

Project Overview

Handwritten character recognition is a field of research in artificial intelligence, computer vision, and pattern recognition. A computer performing handwriting recognition is said to be able to acquire and detect characters in paper documents, pictures, touch-screen devices and other sources and convert them into machine-encoded form. Its application is found in optical character recognition, transcription of handwritten documents into digital documents and more advanced intelligent character recognition systems.

Handwritten character recognition can be thought of as a subset of the image recognition problem. Figure 1 illustrates the general flow of an image recognition algorithm. Basically, the algorithm takes an image (i.e. image of a handwritten digit) as an input, and outputs the likelihood that the image belongs to different classes (i.e. the machine-encoded digits, 1-9).

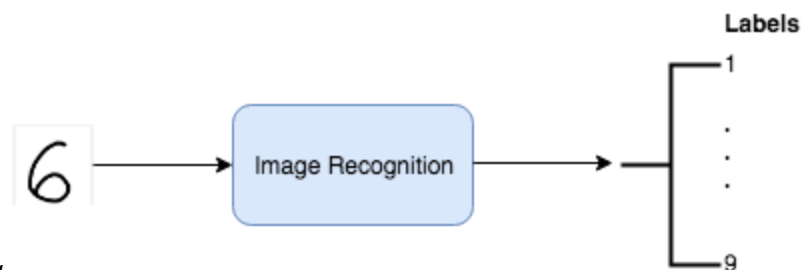


Figure 1

The dataset for this problem will be downloaded from [kaggle](#), which was taken from the famous MNIST (Modified National Institute of Standards and Technology) dataset. Lots of research such as the one done by the AT&T Labs [1] has been conducted using this dataset.

Problem Statement

The problem of this project is to classify handwritten digits. The goal is to take an image of a handwritten digit, and determine what that digit is. The digits range from one (1) through nine (9).

In this study, we will look into the Support Vector Machines (SVMs) and Nearest Neighbor (NN) techniques to solve the problem. The tasks involved are the following:

1. Download the MNIST dataset
2. Preprocess the MNIST dataset
 - a. Apply feature transformation for dimensionality reduction
 - b. Split the dataset into training and test sets
3. Train a classifier that can categorize the handwritten digits
4. Apply the model on the test set and report its accuracy

In section 2 of this study, we describe the image dataset, in section 3 we briefly explain the preprocessing steps of the input data, and describe our implementation of SVM-KNN. In section 4, we present the results of our algorithm.

Metrics

For this project, we will be using the accuracy score to quantify the performance of our model. The accuracy will tell us what percentage of our test data was classified correctly. The accuracy is a good metric choice because it will be easy to compare our model's performance to that of the benchmark as it uses the same metric. Also, our dataset as shown in *Figure 3* is balanced (equal number of training examples for each label) which makes the accuracy appropriate for this problem.

2. Analysis

Data Exploration

In this study, we are using the publicly available MNIST dataset downloaded from [kaggle](https://www.kaggle.com/datasets7634890/mnist-dataset). Some initial data exploration reveals that our training set contains 42,000 samples in total and 784 features. Each sample in the dataset represent an image that is 28 pixels in height and 28 pixels in width, hence the total of 784 pixels. Each image is labeled with their corresponding category that is the actual digit from 0 to 9 for a total of 10 different labels. Some examples are shown in Figure 2

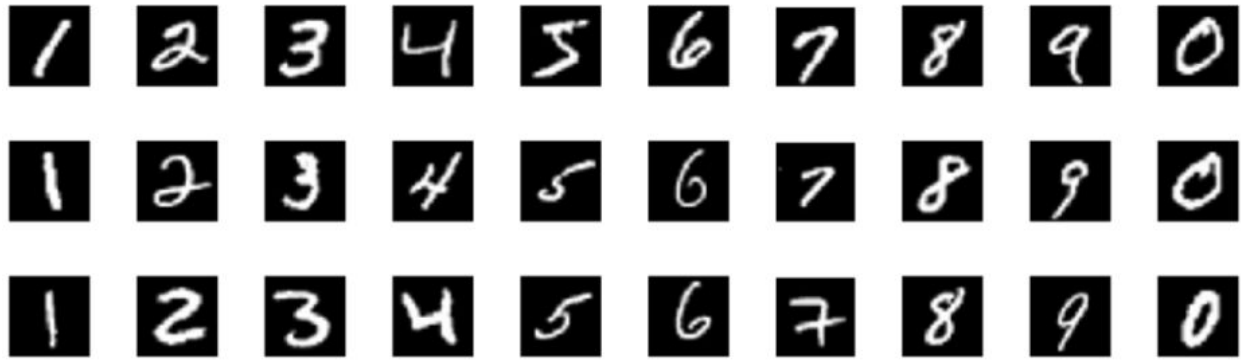


Figure 2

Exploratory Visualization

We have counted the number of occurrences of each label in the training set. Figure 3 illustrates the distribution of these labels. It is obvious from the figure that the distribution is uniform meaning our dataset is balanced.

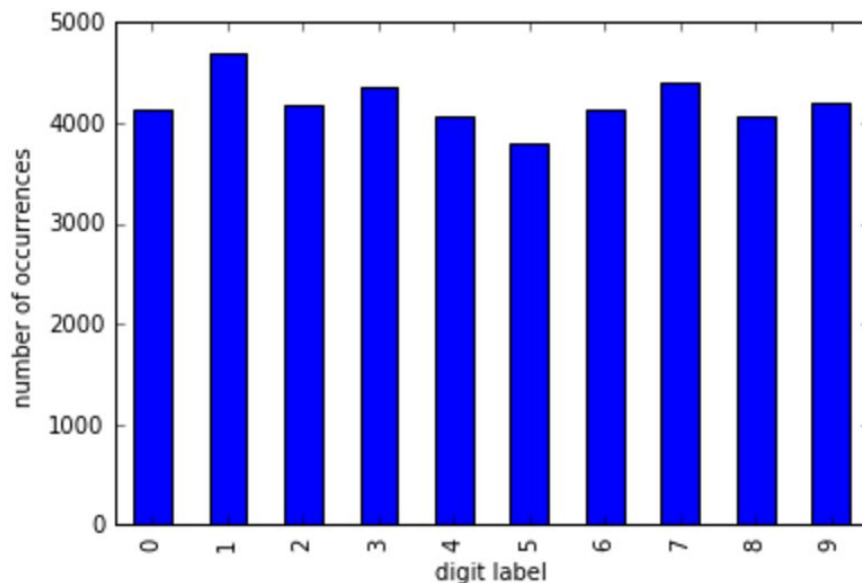


Figure 3

We'd also like to know more about average intensity, that is the average value of a pixel in an image for the different digits. Intuition tells me that the digit "1" will on average have less intensity than say an "8". Figure 4 illustrates the average intensity of each label in the dataset.

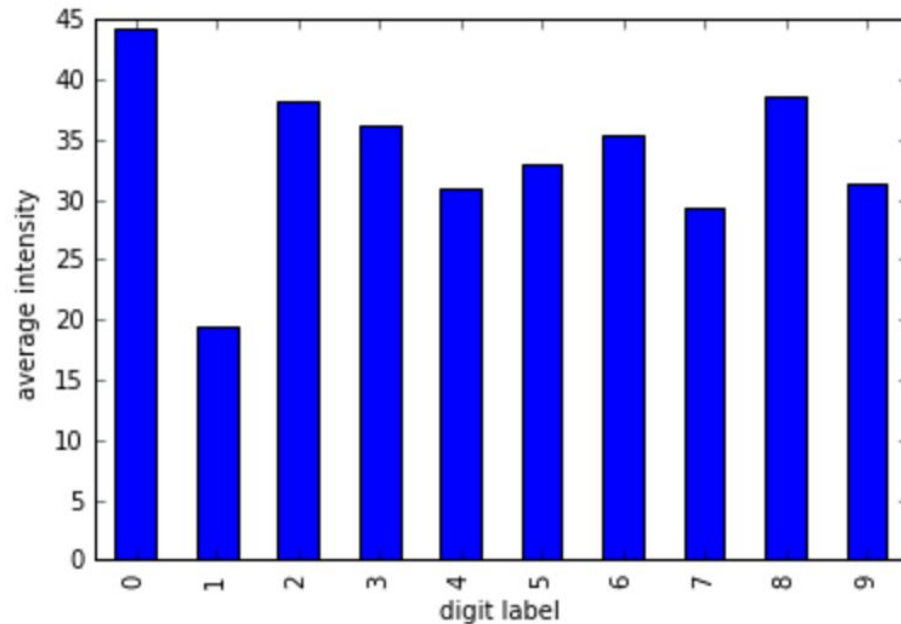


Figure 4

As we can see, there are differences in intensities and our intuition was correct. “8” has a higher intensity than a “1”. Also, “0” has the highest intensity, even higher than “8” which is surprising. This could be attributed to the fact that different people write their digits differently. Calculating the standard deviation of intensities gives a value of **11.08** which shows that there exists some variation in the way the digits are written.

Algorithms and Techniques

It has been shown that Support Vector Machines (SVMs) can be applied to image and hand-written character recognition [4]. SVMs are effective in high dimensional spaces, hence it makes sense to use SVMs for this study given the high dimensionality of our input space, i.e. 784 features. However, SVMs don't perform well in large datasets as the training time becomes cubic in the size of the dataset. This could be an issue as our dataset containing 42,000 samples which is quite large. To deal with this issue, we will adopt a technique proposed by a study conducted at the University of California, Berkeley, which is to train a support vector machine on the collection of nearest neighbors in a solution they called “SVM-KNN” [2]. Training an SVM on the entire data set is slow and the extension of SVM to multiple classes is not as natural as Nearest Neighbor (NN). However, in the neighborhood of a small number of examples and a small number of classes, SVMs often perform better than other classification methods.

We use NN as an initial pruning stage and perform SVM on the smaller but more relevant set of examples that require careful discrimination. This approach reflects the

way humans perform coarse categorization: when presented with an image, human observers can answer coarse queries such as presence or absence of an animal in as little as 150ms, and of course can tell what animal it is given enough time [6]. This process of a quick categorization, followed by successive finer but slower discrimination was the inspiration behind the “SVM-KNN” technique.

Benchmark

To benchmark our model, we will use [kaggle's benchmark model](#) which uses a simple random forest model to make predictions on the test set. The benchmark model accuracy score is **0.93**, in other words it is able to correctly label **93%** of the test data. We will evaluate how this model compares to our final solution. We will aim to have our final solution have an accuracy higher than the benchmark model.

3. Methodology

Data Preprocessing

Since the original dimension is quite large (784 input features), the dimensionality reduction becomes necessary. First we extract the principal components from the original data. We do this by fitting a Principle Component Analysis (PCA) on the training set, then transforming the data using the PCA fit. We used the [PCA](#) module of the [scikit-learn](#) Python library with **n_components** set to **60** to transform the dataset. As shown in Figure 5, the first **60** principal components can interpret approximately 97% of total information (in terms of total variance retained), which suffice to be representative of the information in the original dataset. We thus choose the first 60 principal components as the extracted features.

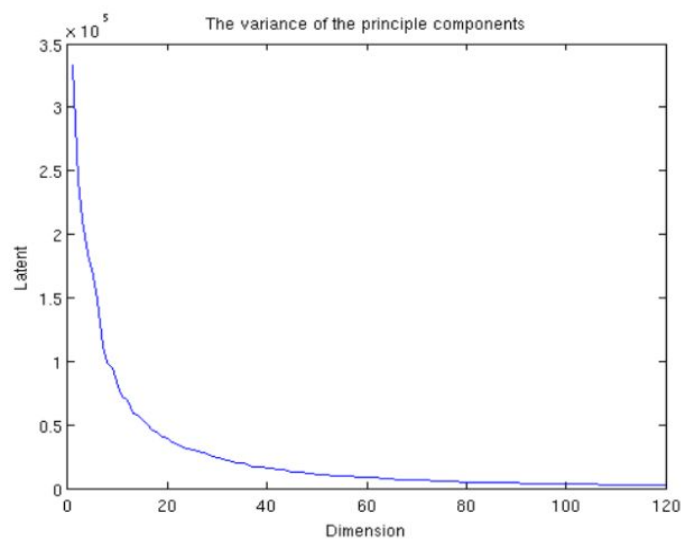


Figure 5

We also applied cross validation to split the dataset into training and testing sets, retaining 40% of the data for testing. We use the StratifiedShuffleSplit module of the scikit-learn Python library passing in the label values as one of the parameters. StratifiedShuffleSplit returns train and test indices to split the data in train and test sets while preserving the percentage of sample of each class.

Implementation

Our simple implementation of SVM-KNN goes as follows: for a query, we compute the euclidean distances of the query to all training examples and pick the K nearest neighbors. If the K neighbors have all the same labels, the query is labeled and exit. Else, we compute the pairwise distances between the K neighbors, convert the distance matrix to a kernel matrix and apply multiclass SVM. We finally use the resulting classifier to label the query. The implementation is illustrated in figure 6.

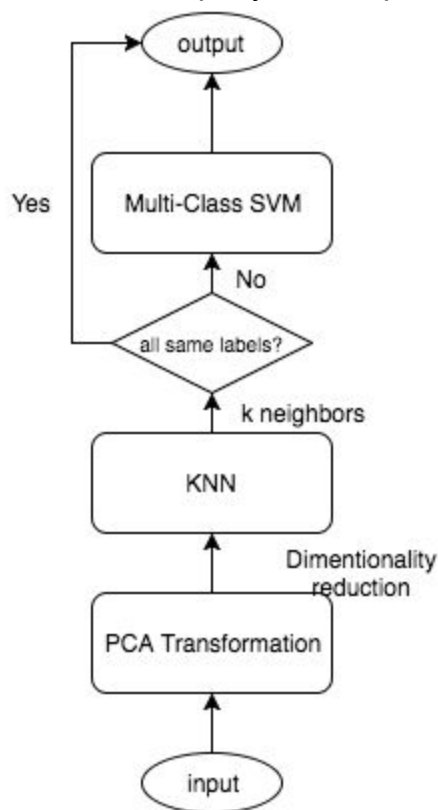


Figure 6

All the algorithms used in our implementation came from the scikit-learn Python library, version 0.17.1. Namely we used PCA for dimensionality reduction, StratifiedShuffleSplit for cross-validation, KNeighborsClassifier to find the k nearest neighbors and SVC to train our multi-class SVM.

Refinement

In our initial implementation, we extract 60 principal components and use parameters values of $k=2$ for KNN and $C=1.0$ for SVM. This resulted in an accuracy score of **0.964**.

k	C	Prediction Time (test)	Accuracy (train)	Accuracy (test)
2	1.0	42.3s	1.0	0.964

I then used a trial and error approach to tune the k parameter by changing its value while keeping the other parameter values the same and observing the results.

- Number of neighbors (k)

k	C	Prediction Time (test)	Accuracy (train)	Accuracy (test)
2	1.0	42.3s	1.0	0.964
3	1.0	50.9s	1.0	0.971
4	1.0	55.6s	1.0	0.969
5	1.0	54.8s	1.0	0.971

- Penalty parameter (C)

k	C	Prediction Time (test)	Accuracy (train)	Accuracy (test)
3	1.0	50.9s	1.0	0.971
3	0.5	47.5s	1.0	0.971
3	0.1	47.6s	1.0	0.971
3	0.01	48.3s	1.0	0.971

From the experiments we can see that the best value of k is **$k=3$** . However, changes in the C value do not impact our final accuracy score. This was a bit

surprising at first, but after thinking about it I feel this is because the input space to the SVM is very small (size 3) and our SVM algorithm can classify the dataset pretty quickly hence changing the parameters does not have much effect on the accuracy. Our final solution then uses **k=3**, **C=0.5**, and yields an accuracy score of **0.971** meaning we are able to accurately recognize close to **97%** of the handwritten digits.

4. Results

Model Evaluation and Validation

During development, a validation set was used to evaluate the model. I split the dataset into training and test sets. The final hyperparameters were chosen because they performed the best amongst the tried combinations. A final value of $k=3$ yielded the best results, as increasing this value resulted in lower accuracy scores but also increased prediction time as it takes more time to find the nearest neighbors. A lower k value makes sense for our model because we are trying to find the few samples where NN has a hard time establishing a decision boundary and apply SVM to perform a more coarse grained classification.

To verify the robustness of the final model, I use a cross validation technique (StratifiedShuffleSplit) on the dataset to ensure that the model generalizes well by using the entire dataset for both training and testing. The model consistently categorized the handwritten characters with a 97% accuracy.

Justification

Applying the SVM-KNN technique on the dataset downloaded from [kaggle](#), I got the following result:

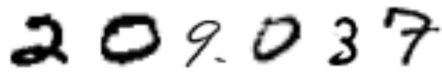
- The classification accuracy is **0.9714**, which is better than that of the benchmark (**0.93514**)

Therefore we can conclude that our model is adequate for solving the problem of classifying handwritten characters in the MNIST dataset as it is able to accurately categorize well with an accuracy quite close to humans. However, our model is useful in a limited domain. Some changes would have to be made to solve a bigger problem of recognizing multiple digits in an image, or recognizing arbitrary multi-digit text in unconstrained natural images.

5. Conclusion

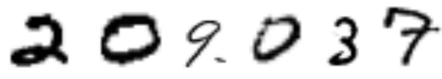
Free-Form Visualization

To test how well our model is working, we attempted to classify the images included in the test dataset provided by kaggle. Here are the actual images and their labels:



2 0 9 0 3 7

These same images were fed to our model, and here is what our model predicted:



2 0 9 9 3 7

Our model incorrectly labels the fourth image and identifies it as a 9, but the correct label is 0. This is interesting because the shape of the top part of number 9 is similar to the shape of 0 which could be the cause of confusion for our model to be able to distinguish the two. One thing is clear that our model could be better at distinguishing shapes more granularly.

Reflection

The process for this project can be summarized using the following steps:

1. Identify an interesting problem and find a public dataset
2. Download & preprocess the data
3. Identify a benchmark for the classifier
4. Research various algorithms and techniques for solving the problem
5. Implement a classifier and train it using the data
6. Refine the classifier until a good set of parameters were found

I found steps 4 and 5 to be the most challenging. I had to familiarize myself with the dataset, search the internet for research papers describing how researchers have approached the problem in the past and understanding the different techniques they used, and finally I had to implement a combination of those techniques.

The most interesting aspect of the project for me was understanding the “SVM-KNN” technique specifically how it relates to the way humans perform coarse categorization in images.

Improvement

One aspect of the implementation that could be improved would be caching the computed pairwise distances when applying SVM on the k nearest neighbors. This follows from the observation that those training examples who participate in the SVM classification lie closely to the decision boundary and are likely to be invoked repeatedly during query time. Adding caching to our implementation would improve the prediction time of our model significantly.

References

- [1] <http://yann.lecun.com/exdb/publis/pdf/matan-90.pdf>
- [2] http://www.vision.caltech.edu/Image_Datasets/Caltech101/nhz_cvpr06.pdf
- [3] http://www.johnwinn.org/Publications/papers/WinnCriminisi_cvpr2006_video.pdf
- [4] <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.441.6897&rep=rep1>
- [5] https://en.wikipedia.org/wiki/Support_vector_machine#Applications
- [6] <https://www.ncbi.nlm.nih.gov/pubmed/8632824>
- [7] https://github.com/chefarov/ocr_mnist/blob/master/papers/knn_MNIST.pdf