Classifying Chinese MNIST Digits

Alex Guo
GitHub repo: https://github.com/ag262/CS532.git

December 12, 2020

1 Introduction

There are many applications for machine learning (ML), but one popular application of ML is in classification of images. For my project, I classified handwritten Chinese digits using three ML algorithms: 1) k nearest neighbors (kNN), 2) support vector machine (SVM), and 3) convolutional neural network (CNN). All of these three algorithms are supervised learning algorithms (i.e. they require labeled training data).

Image datasets are different from most other datasets, since images are extremely unstructured and thus extracting useful/explicit features from images is non-trivial. Consider a black-and-white image of a cat. One method for feature extraction of this image is to flatten the 2D array of pixels into a 1D feature vector, where the intensity of each pixel is a feature; however, one might imagine that a slight translation or rotation of the cat in the image can lead to a very different feature vector. So, preprocessing (such as centering the data) can help dumb algorithms like kNN and SVM do better. Alternatively, one can employ a smarter algorithm like CNN that uses convolution to automatically "filter" the data, thereby making the algorithm more robust in classifying non-preprocessed data. After describing my dataset and parameters used in the three algorithms a bit more, I'll then show the accuracy of each algorithm on both preprocessed and non-preprocessed data.

2 Description of the dataset

The dataset I'm using contains images of handwritten, chinese characters (which represent numbers). There are 15 classes for the data, since there are 15 different numbers ($0 - 10$, 100, 10^3, 10^4, and 10^8) that are each represented with a single chinese character. The dataset contains 15,000, 64×64, black-and-white images of a single character, and there are 1,000 images for each type of character. The link to the data[1] is: https://www.kaggle.com/gpreda/chinese-mnist/notebooks.

3 Description of the three algorithms

3.1 k-NN (k-nearest neighbors)

The k-NN algorithm is very simple. There is no actual training involved, and you can just go right to testing the accuracy of the algorithm on the test data. If you have a training data x, you look at the nearest k neighbors of x and then label x equal to the most popular class of the k neighbor. The two parameters of interest are: 1) k, and 2) the distance metric used to determine who the nearest neighbors are (i.e. L1 or L2 norm).

3.2 SVM (support vector machines)

SVM is a max-margin classifier. One can use the kernel trick (i.e. a non-linear kernel such as the Gaussian kernel) to implicitly map the inputs to a higher dimensional feature space to allow for complex, non-linear decision boundaries. The two parameters of interest are: 1) the type of kernel, and 2) the regularization term λ that places emphasis on how much to maximize the margin.

3.3 CNN (convolutional neural networks)

Neural networks use activation functions to add in non-linearity for classification. CNN involve additional convolution layers that effectively just filter the data before passing the data to a traditional feed-forward neural network. The four parameters of interest are: 1) the convolution section (i.e. number of layers and types of convolution done) 2) the feed-foward section (i.e. number of layers and types of activation functions), 3) number of epochs, 4) batch size (number of samples to use at each gradient update step), and 5) additional tweaks.

4 Methodology

4.1 Pre-processing

I centered and rescaled the images such that there's a set margin around each chinese charater. Then, I applied Otsu thresholding to bring out the foreground of the images and remove any noisy-looking pixels.

4.2 Splitting data and choosing parameters

For all three algorithms, I used 20% of the 15,000 images for testing (i.e. 12,000 for training). For the training data, I did 10-fold validation to ensure accurate errors rates.

For the first two algorithms, I did hyperparameter tuning to choose the ideal parameters (note, I only did this tuning on the preprocessed data). For the second algorithm, I did some coarse tuning without k-fold validation to find the best kernel; then, I did normal k-fold validation to determine the best λ (the reason is that SVM takes a long time to run). For the CNN algorithm, there's simply too many hyperparameters and combinations of them, so I just did some coarse tuning to find what lead to the smallest error rate.

4.3 Comparing algorithms

Let e be the percent error of misclassifiation. I compared algorithms by seeing which had the lowest e when evaluating the model on the testing data. As a baseline, the algorithm is actually learning something if $e < 100 \cdot 10/15 = 93.3$ (which is what you get by randomly guessing).

5 Results and discussion

5.1 Tuning

Table 1 below shows the hyperparameter tuning results on the preprocessed validation data for the first algorithm. Note that the values are the CV errors and thus there're standard deviations associated with them.

Table 1: CV errors for kNN (for tuning)

	$k = 1$	$k = 5$	$k = 7$
L2-norm	14.3 ± 0.8	16.4 ± 0.9	19.4 ± 0.9
L1-norm	14.3 ± 0.8	16.4 ± 0.9	19.4 ± 0.9

Table 1 shows that the distance metric used doesn't matter and $k = 1$ is best. The reason could be that the $64 \cdot 64 = 4096$ dimensional feature vectors are just really "far" away from each other in vector space.

Table 2a below shows the hyperparameter tuning results for finding the best kernel for SVM with λ set to 5 (note that the values are not the CV errors since I didn't do k-fold validation here). Table 2b shows the tuning to find the best λ at the best kernel (the values are CV errors for this table though).

Table 2: Errors for SVM tuning (using $\lambda = 5$)

linear	poly	rbf	sigmoid
11.7	3.2	2.23	26.03

Table 3: CV errors for SVM tuning (using rbf kernel)

$\lambda = 1$	$\lambda = 5$	$\lambda = 10$
4.4 ± 0.4	3.06 ± 0.2	3.05 ± 0.2

The rbf (radial distribution function) kernel is ideal for some reason. The ideal λ being large is due to the fact that the data is not linearly separable and allowing a larger margin with a larger λ allows for more robustness to separating the data.

For the CNN algorithm, I found that using a batch size of 128, 10 epochs was ideal. I used the default learning rate (i.e. factor in front of the gradient term) for Tensorflow (pretty sure the rate just gradually decreases as training goes on. For the convolution section, my input was a size (64,64,1) array of the gray-scale image that was then convolved with a 6×6 filter into a (59,59,60) array that was then max-pooled with a 2×2. For the feed-forward section, the 3D array from before was flattened and fed into a 200-node ReLu layer that then fed into a 15-node softmax layer for giving the outputted class (the softmax applies a probability distribution to determine the class). The dropout rate was set to 0.3, which gives the probability to drop/ignore a node each time we backpropagate (this helps prevent overfitting).

5.2 Evaluating/comparing the algorithms

Table 3 shows the errors (e from evaluating the test data) of each algorithm at their ideal tuning parameters when the 15,000 images were preprocessed vs. non-preprocessed.

Table 4: Errors of algorithms at their ideal parameters

	kNN	SVM	CNN
Preprocessed	13.5	3.1	1.6
Non-preprocessed	65.7	21.8	4.9

For the first two algorithms, the testing error rate is close to the CV error from tuning, meaning 10-fold validation helped prevent overfitting. kNN performs the worst, since the algorithm is very simple. SVM (with the rbf kernel) performs the second worst. However, it is clear that kNN and SVM gain more (in terms of how much e is lowered) from using the preprocessed data due to the fact that giving more consistency between feature vectors helps counteract slight shifts/rotations of the handwritten characters.

Although one might say CNN does the best because it's the most complex algorithm, I'd argue CNN is fundamentally very simple compared to the other two – CNN just relies on convoluted convolution to do some extra filtering and the feed-forward part just does some simple layer-weight multiplications. If you add preprocessing of the data on top of what CNN already does, you gain a lot of robustness when it comes to classifying image data (thus why e is the smallest).

6 Conclusion

15,000 images of 15 types of Chinese characters were classified. Since image data is very unstructured, I found that preprocessing is the most important step that enables successful classification of the data. The next most important step is choosing a good algorithm. For the preprocessed data, the misclassification error percentage e was 1.6 for CNN, 3.1 for SVM, and 13.5 for kNN. Since the convolution layers in the CNN algorithm offer additional filtering properties on top of the preprocessing, CNN allows for more robustness to slight translations/rotations in the image data. So, if you want to classify images, preprocessing and CNN are the way to go.

References

[1] Chinese MNIST. (n.d.). Retrieved December 12, 2020, from https://kaggle.com/gpreda/chinese-mnist