Digit Recognition Using KNN and SVM

Harnoor Singh
Department of Computer Science & Engineering
University of Washington
hsingh@cs.washington.edu

Brian Walker
Department of Computer Science & Engineering
University of Washington
bdwalker@cs.washington.edu

1 Objective

As described in our project proposal, we are using the Kaggle digit dataset to classify written numbers.

Our goal is to implement digit recognition using K Nearest Neighbors (KNN) and a Support Vector Machine (SVM) and compare the two approaches. A stretch objective is to enable data that isn't a part of the Kaggle dataset to work with our algorithms. More specifically, we would like to be able to classify some basic mathematical symbols in addition to digits. This would allow us to be able to classify and evaluate simple mathematical expressions. There is some challenges associated with this. We would need to create our own data set as well as normalize the input to match with the Kaggle input. Thus, this is a stretch goal that we hope to get to but only if time allows.

2 KNN

2.1 KNN Progress

We have made significant progress in both the KNN algorithm and the SVM algorithm. Our KNN algorithm is completely implemented k-fold cross validation to tune our parameter K.

The Kaggle dataset provides a larger training and testing set. However, we do not have the gold standard labels for the testing data. In order to evaluate the accuracy we have to submit our output into the Kaggle Digit Recognition competition. Our implementation did surprisingly well, placing us at 263rd place out of 1237 with a successful classification rate of .97114. This error rate was achieved with a 1NN implementation. Since then, we have reclassified and resubmitted with 15NN, which dropped our accuracy to .963. We are quite happy with the results for such a simple algorithm. After performing our cross-validation on a small subset, it would appear that 1NN may be the best choice for K. If this is the case then there is little optimization we can do to improve our implementation. We are hoping the flexibility of a kerneled SVM will allow us to improve our classification.

As we discuss below, we will run cross-validation on the entire data set for the final report to ensure we have chosen the best candidate for K. We will also run some more classification runs on the test data. The current classification takes around 7.5 hours on my home computer which has 16Gb of RAM and an Intel Extreme Core I7 processor. We will implement a threading procedure to speed up the classification before the final report to allow us to run more testing.

2.2 KNN Procedure and Optimization

To compute the K nearest neighbors, we take the euclidean difference between the feature we are classifying, x_1, and every feature in our training set.

$$\sum_{i=1}^{D} (x_{1i} - x_{2i})^2$$

Where D is the number of features for a sample.

Once we have calculated the K closest features, a simple voting method is used to classify X_1.

In order to select a value of K we do k-fold cross validation on a subset of the training set. The data below was sampled using 1000 samples. We will perform cross validation with a larger data set for the final report but due to time constraints and the time required to do k-fold validation on the dataset provided by Kaggle, we limited the number of samples for this report.

We performed the k-fold validation by splitting the training data into k evenly divided sets. We then use k-1 of these as "training" data and classify the remaining fold using the k-1 other sets. We then rotate through the subsets and test with each one. We then average the results once testing has been done with each of the subsets. We do this for different values of KNN and pick a value of K that gives us the best results.

The K value that led to the lowest error rate is used to build our final classifier. The result of our cross validation can be seen in Table 1 below.

Table 1: Cross Validation Selection of K

K	Misclassified
1	135
2	164
3	137
4	148
5	144
10	163
15	176
20	191

There is a large disconnect between our full classification error as obtained from Kaggle and the error rate above. This can be attributed to a much lower number of training samples.

3 SVM

3.1 SVM Progress

We have made good progress on our kerneled SVM implementation. We are working on some issues with performing regularization and tuning the additional parameters we need for SVM. We

our currently capable of running the SVM with the digit recognition data but we are not confident in the results. We suspect we have a day or two worth of work to get the SVM completely running.

The implementation has been designed such that we can plug in different kernels for easy testing. These kernels are also "curried" to allow them to be easily instantiated with their own tuning parameters for passing to the SVM. Thus we can easily optimize our classifier by performing cross-validation with both different kernels and different parameters to the kernels themselves. It is our hope that this flexibility will allow us to classify the trickier digits in the competition.

3.2 SVM Procedure and Optimization

The SVM implementations we have worked with previously have involved only two classes. Because we have to classify the digits 0-9, we need to create an SVM that can classify from a broader range. The two primary methods for a multivariate SVM classifier is a one vs. one or one vs. all implementation. There has been some debate about which is better and there does not appear to be a clear winner from the research we have done. We have chosen to implement the one vs. one method for out multivariate SVM.

For one vs. one we must train a classifier for every pair of possible classifications. Since SVM is a binary classifier we must consider every possible combination. Thus, we must train $nchoose2$ SVMs which compare two numbers. For example, one SVM would classify a digit as either a 1 or a 2. Another would classify a digit as a 1 or a 3, and so on. We get a prediction from all of these classifiers and take a vote to see which digit was most likely based on our SVMs.

In our training step, every time we make a mistake we log it. Future predictions are made based on past mistakes.

$$Classification = sign(\sum_{i \in Mistakes} \alpha_i (Y_i * k(X_i, X)) + w_0)$$

A classification is considered a mistake if the truth label y multiplied by our prediction is less than 0.

The biggest challenge thus far has been deciding how to regularize our SVM without relying on the weights. Because we use the kernel trick, the weights are never explicitly stored.

4 Future Work

Our primary focus moving forward is to get our SVM implementation completed. Most of the structure is in place, however we need to decide on our method of regularization. Once we have the regularization done we can analyze the effectiveness of our SVM. We will implement cross-validation to tune the regularization as well as the kernel and the kernel dimensions. The SVM provides much more flexibility through parameters than KNN does and thus cross-validation will be very important to ensure that our classifier is effective.

For KNN it might be beneficial to attempt to increase performance. Classifying each sample takes about 1 second, and with the testing dataset holding about 280,000 samples it takes several hours to classify all the points. Possible areas of speedup are a better algorithm for finding the nearest neighbors or parallelizing the nearest neighbor search.

If we have enough time we will then attempt at making our own dataset for classifying mathematical symbols. We have been unable to find a dataset that provides ground truth labels to symbol training data. If we cannot find a suitable provider for the data we will have to make our own. We suspect that this will not be very accurate since we will only be able to make a limited number of samples. However, it is something we would like to explore.