Digit Recognition: Nearest Neighbors vs. Perceptron

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

Abstract

Optical Character Recognition (OCR) is a complex computer science problem which has several effective machine learning solutions. We implemented the K Nearest Neighbors (KNN) and Kernelized Perceptron algorithms and compared the effectiveness of both methods in classifying a testing data set of 28,000 images. Each image is formatted as an array of 768 pixel values, with each value indicating how dark each pixel was in the image. Both of our algorithms were trained on a training data set of 42,000 images. We used cross validation to tune the K value in KNN and to choose a kernel function and dimension value for Perceptron. Despite its relative simplicity, we found that KNN was more effective than Perceptron in classifying the digits. The nearest neighbors algorithm correctly classified 96.857% of the testing samples on a weighted neighbors algorithm which looked at 4 neighbors. Our best Perceptron implementation correctly classifies 96.3% of the testing samples using an exponential kernel with a $\sigma = 10$. Despite its higher accuracy, the runtime of KNN was 8x slower than that of the Perceptron.

1 The Problem

1.1 Image Data

Optical Character Recognition (OCR) is a computational problem where a machine attempts to classify an image of text. We tackled a subset of this problem, attempting to classify the digits 0-9 from data provided by the Kaggle Digit Recognition competition[1].

The Kaggle data provides two groups of data, 42,000 training samples with labels and 28,000 unlabeled testing samples. Each training sample contains a 48 pixel by 48 pixel image containing a handwritten number. The image is represented as an array of 768 pixel values which represent how dark each pixel of the image is. Table 1 is an example of a training data sample.

[1]http://www.kaggle.com/c/digit-recognizer

Table 1: Subset of Kaggle Data

Label	Pixel 77	Pixel 78	Pixel 79
8	0	0	0
6	89	208	135

2 K Nearest Neighbors (KNN)

2.1 Overview

K Nearest Neighbors is a relatively simple algorithm which often gets surprisingly good results given its simplicity. Given a testing sample X, the algorithm looks through all the training samples it has previously seen and finds the *K* most similar samples, known as the nearest neighbors. The algorithm then implements some form of a voting mechanism among the *K* samples to classify the testing sample as a digit 0-9. The voting mechanism can be as straightforward as selecting the majority label, or it can be more complex, involving weighting samples based on some distance metric.

2.2 Implementation Details

The first step in implementing KNN is to find the *K* nearest neighbors. This is accomplished by taking the euclidean distance between the testing sample we are classifying and every training sample in our training data set.

$$Euclidean(TestX, TrainX) = \sum_{i \in TrainX} (TestX - TrainX_i)^2$$

Because the image data is defined as an array of 768 pixels, $TestX - TrainX_i$ is actually doing a 768 dimension comparison between data points.

Once we identified the *K* nearest neighbors to our testing sample, we had to use this information to classify the testing point. We considered two approaches when deciding how to perform this classification.

2.2.1 Unweighted KNN

The first method we implemented to classify our testing sample was a simple majority voting algorithm which selects the majority label from the *K* closest neighbors. For example, if we ran the nearest neighbors algorithm with $K = 5$ and 3 of the neighbors had a label of 6, we would classify our testing sample as a 6. If there is a tie between two labels then the label is picked arbitrarily.

2.2.2 Weighted KNN

Given the *K* nearest neighbors, the weighted version of the algorithm multiplies each label by a weight which indicates how close it is to our testing sample. These weights help to emphasize points that are closer to our testing sample while reducing the impact further away training points have on classifying our testing sample.

Many different weight functions are available but we chose to use the inverse distance. This allows samples that are closest to one another to have the largest impact while discounting those that are farthest away. The distance formula we used is below:

$$Weight(TrainY, TrainX, TestX) = \frac{1}{Euclidean(TestX, TrainX)} * TrainY$$

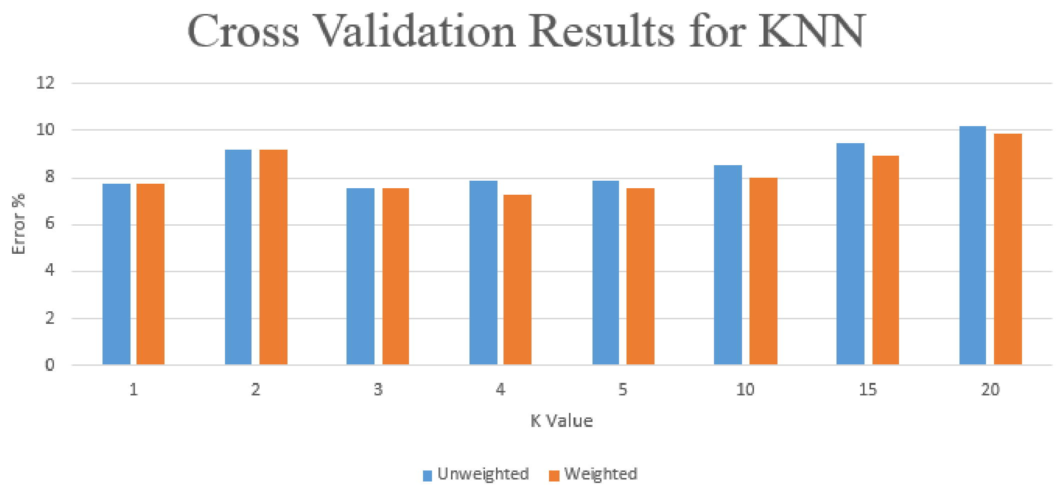
2.3 Cross Validation

Our implementation of KNN has a number of parameters which require tuning. In particular, we need to decide whether or not to use the weighted version of our KNN algorithm and how many neighbors to consider when implementing our algorithm.

To do this, we implemented K folds cross validation using 5,000 training samples. The reduced training set allowed us to run our algorithm using various configurations to see which configuration was most accurate.

Table 2: KNN Cross Validation Error Rates

K	Unweighted KNN Error	Weighted KNN Error
1	7.7%	7.7%
2	9.2%	9.2%
3	7.5%	7.5%
4	7.8%	7.3%
5	7.8%	7.5%
10	8.5%	7.9%
15	9.5%	8.9%
20	10.2%	9.8%



Our cross validation results suggest that running weighted KNN with $K = 4$ yields the most successful classification rate. One caveat to this is that our cross validation code ran using 5,000 training samples instead of the 42,000 samples in our full training set. It is possible that data we used to cross validate is not representative of the full data set.

2.4 Results

Our KNN implementation was extremely successful, correctly classifying 96.857% of the testing samples. One downside to it, however, is the runtime. Training is near instantaneous since the algorithm simply loads the data into an array. To classify a sample the algorithm needs to compute the euclidean distance between the testing sample and every training sample. In the case of the full data set this is 42,000 samples, each of which has 768 dimensions. As a result, each testing sample takes approximately one second to run. Altogether it takes close to 8 hours to classify all 28,000 testing samples.

2.5 Analyzation

The KNN algorithm is often very accurate but can be too slow for some applications. In the case of OCR it proved to be the easiest to implement and the most accurate. The algorithm uses a brute force tactic to compare a point against all other training points. This allows a decision to be made with all the information available. However, we were still only able to obtain a max of 96.857% accuracy. There are several possible reasons for this. First, OCR is very difficult. There are likely some hand written samples that simply do not have a good representation of nearest neighbors. Peoples handwriting styles are different enough that some of the more unique interpretations of numbers did not get classified. Second, we only ran cross validation on a small subset of the samples. We did not have the time to run it on the full 42,000 samples and thus the k value we used for testing may not have been optimal.

3 Kernelized Perceptron

3.1 Overview

The perceptron algorithm is a binary classifier that minimizes the hinge loss objective function. Typically, this would require storing weights to use in making a prediction during testing. However, the kerneled version of the perceptron instead stores the mistakes made during training to use during testing. All the mistakes made during training are then passed to a kernel with the testing data and test time in order to make a prediction of the testing label.

In its most basic implementation, the perceptron algorithm divides the data linearly. However, the data is often not linearly separable. To address this issue we use a kernel function to create a non-linear decision boundary. This allows us to have a more complex decision boundary which potentially fits the data better.

3.2 Implementation Details

Before we can begin classifying test points, we need to train the perceptron algorithm using our training data set. The training process attempts to classify each training point sequentially. If the perceptron correctly classifies the point nothing changes and the training process moves on to the next training sample. If the training sample is incorrectly classified, however, the mistake is added to an array of mistakes which is taken into consideration for future classifications.

One problem that needed to be addressed is that a standard perceptron is a binary classifier meaning it selects between one of two classes. We resolved this issue by training $\binom{10}{2} = 45$ classifiers. Each classifier is responsible for classifying between two digits. For example, we would train a perceptron to classify between 0 and 1, and a second to classify between 0 and 2, and so on. When we classify we vote between all of our trained classifiers and output the majority element.

The actual classification is performed by:

$$classify(X) = \sum_{i \in Mistakes} Y_i K(X_i, X)$$

The kernel function is represented by $K(X_i, X)$. There are a number of choices for this kernel function which we will explore in the next section.

3.3 Cross Validation for Kernel Selection

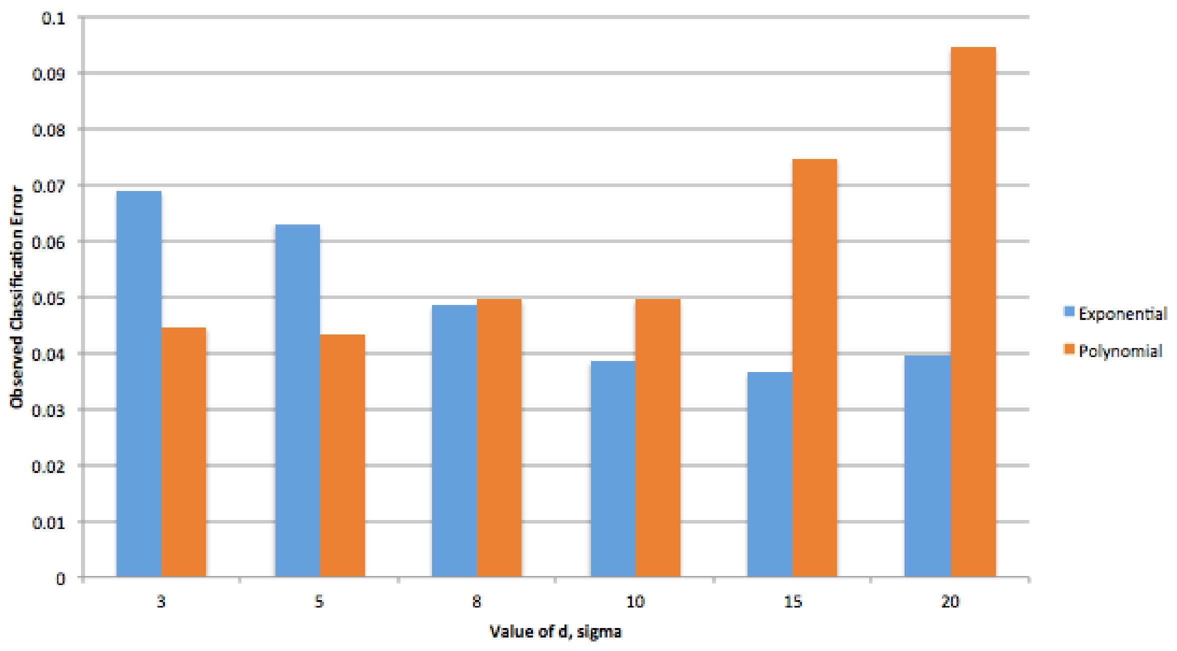
We implemented two kernel functions to use with our perceptron algorithm. One is a polynomial kernel which takes the form:

$$Polynomial(x, y, d) = (dot(x, y) + 1)^d$$

The second is an exponential kernel:

$$Exponential(x, y, \sigma) = exp(-\frac{||x-y||}{2\sigma^2})$$

Both of these kernels have a parameter which needs to be tuned, d in the case of the polynomial kernel and σ in the case of the exponential kernel. We used 10-fold cross validation to find out which kernel gave us the best result as well as which value of d and σ was optimal. Below are the results of the cross validation:



There are many different possible selections of kernels to use and our implementation allows for the user to pass a command line argument to select the kernel and dimension. We chose these two because we knew the performance was adequate based on our previous homework. Due to time constraints we were not able to cross validate and expire any other kernels. We briefly explore the Gaussian but the mistake files grew to over 1Gb in size and thus the training slowed considerably.

3.4 Results

Our cross validation indicates that the exponential kernel with a σ of 15 leads to the highest success rate of 96.3%. An interesting observation is that our polynomial kernel actually became less successful when it's dimension was increased whereas the exponential kernel got better. We suspect that this is because the polynomial kernel began to overfit the training data.

Finally, the runtime of the perceptron algorithm is a little over an hour. We timed a particular train and test run with the exponential kernel and found that it trained in 21 minutes and tested in 48 minutes. This was similar to what we observers with the polynomial kernel as well. The faster runtime allowed us to cross validate on the full data set so our cross validation results are more accurate than those we obtained for KNN.

4 Conclusion

Despite its seeming simplicity, we found that KNN was extremely successful when classifying data points in the Kaggle set.

Table 3: KNN vs. Perceptron on Full Testing Set

Algorithm	Percent Success	runtime
Nearest Neighbors, $k = 4$	96.857%	$\sim$8 hours
Perceptron	96.129%	$\sim$1 hour

Table 4 summarizes the testing accuracy and runtimes of weighted KNN with $K = 4$ and perceptron with the exponential kernel and $\sigma = 15$. KNN is clearly more accurate, however because perceptron is nearly 8x faster, there could be many cases where it is a better choice.

In addition to being faster, perceptron likely has a lower memory footprint. While KNN must store every sample in the training set, perceptron only stores mistakes which will be a subset of the training set.

One reason why KNN may have been so successful in this trial is due to the large training data set and small number of classes to classify. With only 10 possible classifications, and 42,000 training samples, the odds of at least a few of those being very similar to the test image intuitively seem high. As briefly mentioned earlier there will be a small minority of strangely written numbers that will not have a good match to any of the training samples and thus will not be classified properly.

An explanation for the perceptron algorithm being less successful than KNN is that it may have overfit our training data because there was no regularization. This could explain why our polynomial kernel did worse with a higher dimension kernel function. We would have liked to explore a kerneled SVM to resolve this issue but we ran into problems with derivation and implementation. Due to time constraints we stuck with the perceptron and explored its implementation further through cross validation and kernel implemenations.

5 Future Work

There are three specific expansions to our work we would have liked to explore had we had time.

- Expanded cross validation for KNN using the full training set. This would help validate that the weighted voting algorithm with $K = 4$ leads to the most accurate classification.
- More kernels and a greater assortment of σ and d values to see if we can increase the effectiveness of the perceptron algorithm.
- An SVM implementation which regularizes the data to avoid overfitting. If the perceptron implementation overfits the data, this would be a potential way to increase the accuracy.