An Investigation of Classification Techniques For Handwritten Digit Recognition

Deric Pang
dericp@cs.washington.edu

Saidutt Nimmagadda
nimmas@cs.washington.edu

Abstract

TODO

1 Introduction

TODO

2 Project Description

2.1 Goal

Our goal for this project is quite simple—we want to take pixel data from images of hand-drawn digits and classify them as a number from 0 to 9.

2.2 Data

The data for our project was taken from the MNIST dataset. As written on a Kaggle competition using the MNIST dataset, "The MNIST ('Modified National Institute of Standards and Technology') dataset is a classic within the Machine Learning community that has been extensively studied. More detail about the dataset, including Machine Learning algorithms that have been tried on it and their levels of success, can be found at http://yann.lecun.com/exdb/mnist/index.html."

Each handwritten digit is vectorized. Each vector is composed of a label which represents the label classification (i.e. numbers from 0 to 9) and 783 pixel features of integer values between 0 and 255. Since all of the data is already normalized and centered, we did not need to take those steps ourselves to start building models against. The training data set consists of 60,000 data points, while the test set consists of 10,000 data points.

The specific flavor of the MNIST dataset that we used came from https://pjreddie.com/projects/mnist-in-csv/ where the data is formated as CSV files.

3 First Considerations

3.1 Machine Learning Techniques

TODO

3.2 Related Work

Before we began implementing any classifiers, we wanted to investigate what material already existed in the domain of digit recognition. We discovered that digit recognition, especially on the

MNIST database, is an extremely well studied problem. As such, we had no trouble finding a plethora of detailed papers.

Going into the project, we had intuition that the most accurate classifier we could build was a convolutional neural network. This was confirmed in most of the related work [1 - 3]. However, we wanted to determine which techniques would be the most insightful, interesting, and performant to implement for our project even though the readings pointed out a clear state-of-the-art.

We saw from the conclusions of Lecun [1] that a k-Nearest Neighbors classifier would not only pose serious scaleability difficulties when it came to runtime and memory usage, but it would also be a comparatively unreliable classifier. We decided that it would be interesting to see just how difficult it would be to deal with the runtime and what kind of accuracy we could achieve with limited computation resources.

We also saw that according to Maji [2], "with improved features a low complexity classifier, in particular an additive-kernel SVM, can achieve state of the art performance." This inspired us to implement a support vector machine as a lightweight, high accuracy classifier.

4 Approach

4.1 Baselines

We used scikit-learn [6] to implement baselines for the k-NN classifier and the SVM. Even though scikit-learn is presumably optimized very well, these classifiers took a very long time to train and evaluate. As a result, we tuned hyperparameters with only 17% of our training data. Once we settled upon optimal hyper parameters, we trained the scikit-learn k-NN and SVM on 25% of the training set and evaluated them on the entirety of the test set.

4.2 Evaluation

To tune hyperparameters, we used cross-validation with random 80-20 splits. We consistently report the validation errors from the cross-validation and the test error on the MNIST test set.

4.3 k-Nearest Neighbors (k-NN)

4.3.1 Scikit-learn k-NN

To baseline our k-NN classifier, we used scikit-learn's k-NN implementation. Scikit-learn's k-NN can utilize different approaches like brute-force, a K-D tree, and a Ball tree. The K-D tree and Ball tree trade precision for efficiency by utilizing heuristics to prune which points are considered when evaluating the nearest neighbors. Scikit-learn decides which algorithm to use based on the number of samples N, the dimensionality of the data D, the structure of the data (e.g. sparsity, intrinsic dimensionality), the number of neighbors k requested for a query point, and the number of query points.

After tuning k using cross-validation, we found that scikit-learn prefers to use a K-D tree. Running cross-validation over a set of k values between 1 and 625, we found that $k = 1$ minimized our validation error. The results of this cross-validation can be seen in figure 1.

4.3.2 k-NN Regression

We have implemented a k-NN classifier using k-NN regression. We use a brute force algorithm that scans every point in the training set to find the k neighbors with the lowest Euclidean distances from the query point. Then, amongst those k neighbors, we find the classification with the lowest average Euclidean distance from the query point and assign its value to the query point.

We found the runtime of k-NN when using the entire training set of 60,000 images to be intractable. Given runtime constraints, we used 10% of the training set in our k-NN regression. We found that with $k = 1$, our k-NN classifier obtained the lowest validation error. Although this was consistent with the result of cross-validation with scikit-learn's k-NN classifier, we predicted that with the use of a kernel, our results would change. We hoped that as k increased, our classification error

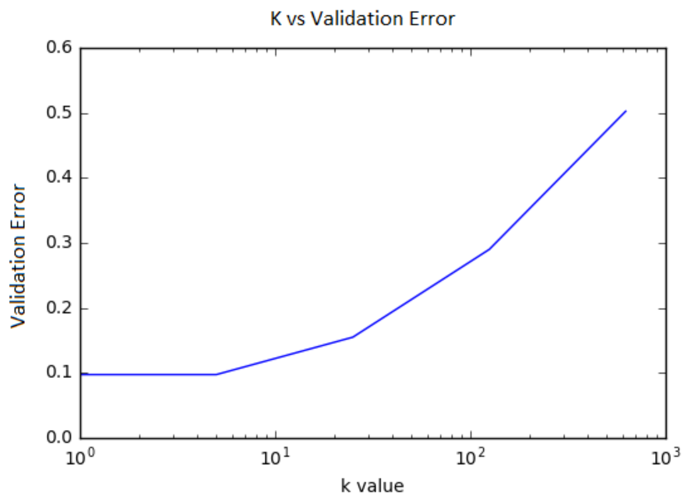

Figure 1: cross validation of scikit-learn's k-NN over varying k values

would decrease. However, without a proper kernel to weigh the data points, we decided that it was possible for larger values of k—analogous to larger bandwidths in the kernelized k-NN—to produce inaccurate classifications. Results of cross-validation over varying C values can be seen in figure 2.

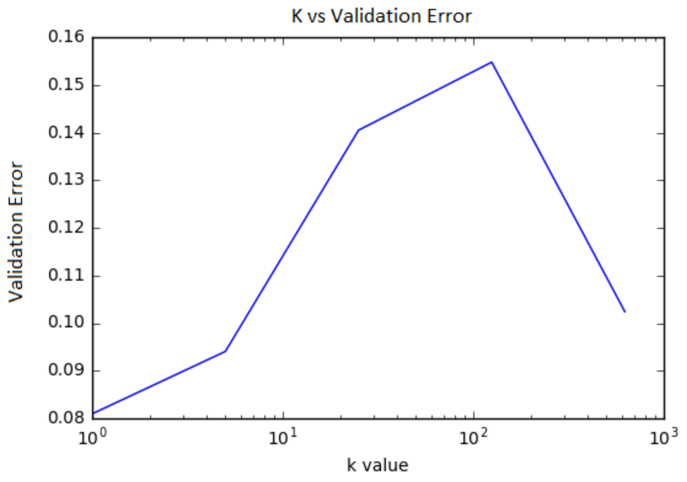

Figure 2: cross validation of k-NN over varying k values

4.3.3 Kernelized k-NN

We implemented a Gaussian kernel and classified query points based off of the Nadaraya Watson Kernel Weighted Average. Instead of finding the k nearest neighbors to classify a query point, we used this Kernelized Regression to calculate the classification for each query point.

TODO include kernel formulation here

Since we didnt have to just find the k nearest neighbors, but instead iterated through every point and kernelized its distance as a weight to its classification, we had to tune our bandwidth value λ rather than a k. As a result of overflow and underflow errors, we could only test a range of bandwidths from 10^4 to 10^6.

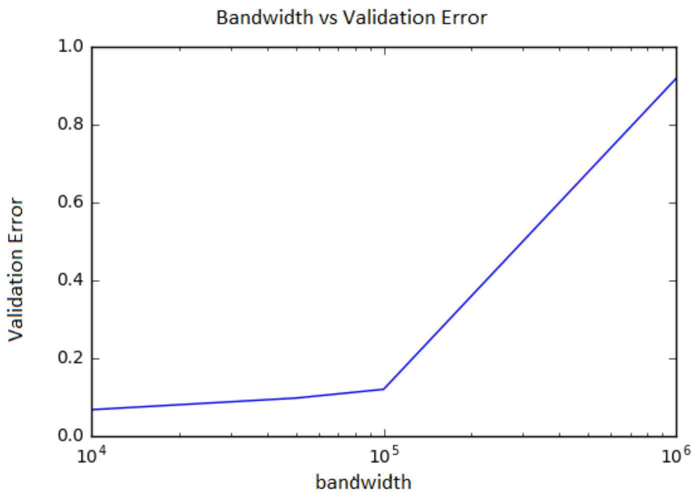

Figure 3: cross validation of kernelized k-NN over varying bandwidth values

Running cross validation on 1/10th of our training set, we found that a bandwidth of 10^4 not only avoided overflow errors, it also minimized validation error.

4.4 Support Vector Machine (SVM)

4.4.1 Scikit-learn SVM

To baseline our SVM, we used scikit-learn's LinearSVC and SVC functionality. The former trains a linear SVM until convergence, and the latter uses a Radial Basis function kernel.

We performed cross-validation over varying C values on 17% of the training set (10000 images). We found that C values of 1 and 100 minimized validation error in the Linear and Kernelized SVMs, respectively. This result was surprising to us—in particular, we did not expect a C value of 1 to produce a good result. After further investigation, we suspect that the differences in validation error can be ascribed to differences in validation data splits. Nonetheless, we used those respective C-values for creating our model against the the training data and running it against the test data. Results of the cross-validation can be seen in figures 4 and 5.

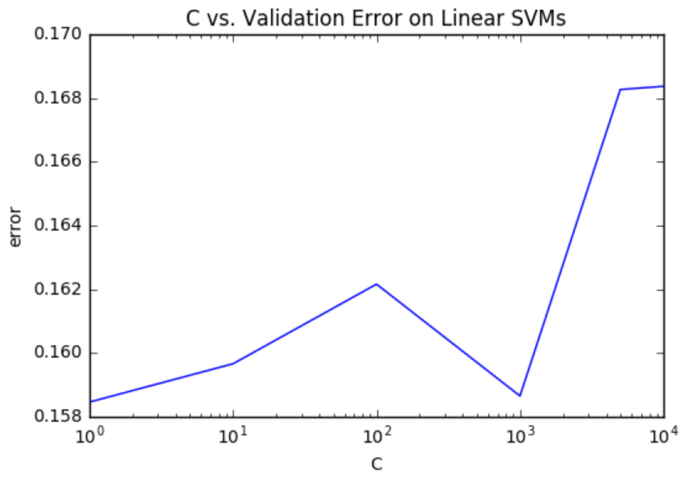

Figure 4: cross validation of scikit-learn's linear SVM over varying C values

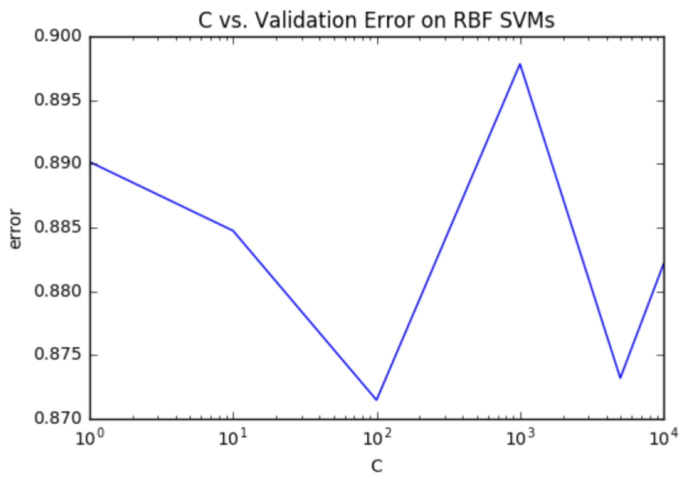

Figure 5: cross validation of scikit-learn's kernelized SVM over varying C values

4.4.2 SVM Trained With Stochastic Gradient Descent

We have implemented a vanilla SVM trained with SGD. Performing cross-validation over varying step-sizes and C values, we obtained an optimal step size of $1 * 10^{-11}$ and an optimal C value of $12,000$. Results of the cross-validation can be seen in figure 6.

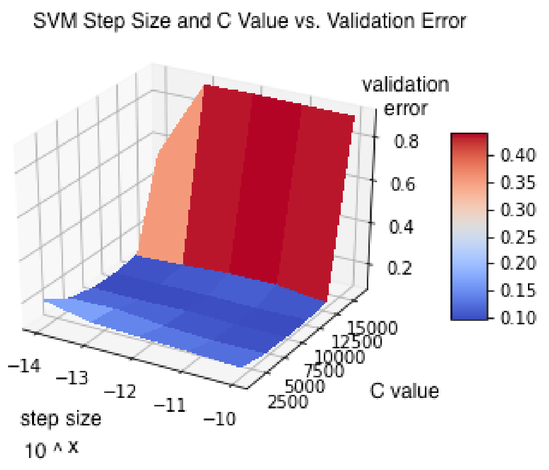

Figure 6: cross validation of SGD SVM over varying step sizes and C values

This SVM was fast to train and performed quite well as we will see in section 5.

4.4.3 SVM Trained With PEGASOS

We decided to investigate different training methods for SVMs. As a result, we also trained an SVM with the PEGASOS [4] algorithm. Performing cross-validation over varying λ values, we obtained an optimal λ value of 10^{-7}. Results of the cross-validation can be see in figure 7.

4.5 CNN

TODO

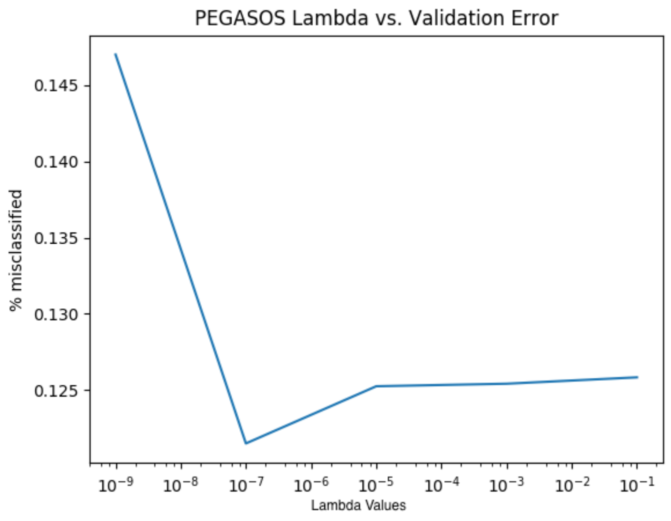

Figure 7: cross validation of PEGASOS SVM over varying λ values

5 Results

References

[1] LeCun, Yann, et al. ”Comparison of learning algorithms for handwritten digit recognition.” International conference on artificial neural networks. Vol. 60. 1995.

[2] Maji, Subhransu, and Jitendra Malik. ”Fast and accurate digit classification.” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-159 (2009).

[3] Sundaresan, Vishnu, and Jasper Lin. ”Recognizing Handwritten Digits and Characters.” (1998).

[4] Shalev-Shwartz, Shai, Yoram Singer, and Nathan Srebro. ”Pegasos: Primal estimated sub-gradient solver for svm.” Proceedings of the 24th international conference on Machine learning. ACM, 2007.

[5] Rahimi, Ali, and Benjamin Recht. ”Random Features for Large-Scale Kernel Machines.” NIPS. Vol. 3. No. 4. 2007.

[6] Pedregosa, Fabian, et al. ”Scikit-learn: Machine learning in Python.” Journal of Machine Learning Research 12.Oct (2011): 2825-2830.