**1 Project Description**

In our project, we implemented different algorithms designed to classify handwritten digit data into their appropriate digit classifications. Our data set is the MNIST handwritten digits dataset that consists of one training set and one test set. Each data point was a vector with one value representing the label classification (i.e. numbers from 0 to 9) and 783 pixel features. Each pixel feature consisted of a value 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 consisted of 60000 data points, while the test set consisted of 10000 data points. Our goal with this project was to evaluate the performance of three different algorithms, two of which we’d implemented from scratch and refined with different techniques, on our test set.

**2 Approach**

Our approach was to first decide which algorithms we would use. Then, since we wanted a reference to compare our results to, we made baseline implementations of the algorithms we’d implement (Nearest Neighbors, Support Vector Machines). After making those baseline implementations, we ran cross validation on 1/6 of our dataset in order to tune hyperparameters. Once we settled upon optimal hyper parameters through that cross validation, we used those hyper parameters to train our model against 25% of the training set and test that built model against our entire test set. We chose to train our model against just 25% of the training set because of serious runtime constraints – even from the baselines.

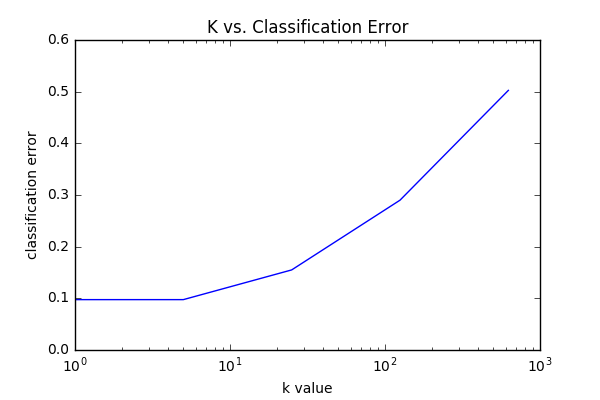
After implementing the baselines, we implemented the algorithms themselves. For each high level algorithm, we implemented different refinements and ran cross validation in order to get a diverse array of data for a diverse array of techniques. We kept a high scope by having multiple algorithms, and we kept a high depth by tinkering with each algorithm.

Finally, we used TensorFlow to implement a convolutional neural network. We knew from the readings we’d read that a CNN would perform optimally, and we wanted to have our own optimal baseline to compare our results to.

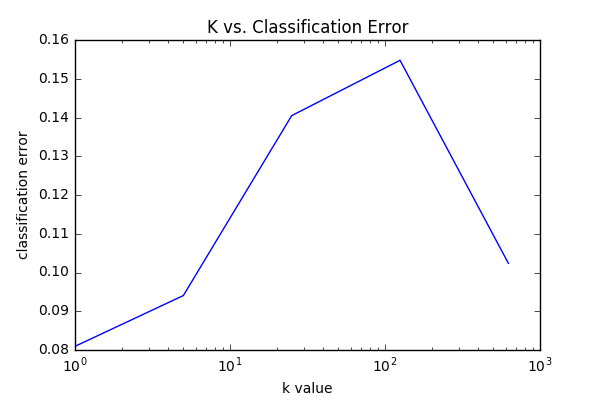
**3 Algorithms**

3.1 Nearest Neighbors Regression

We first tackled nearest neighbor regression. For baselining, we used the scikit-learn NearestNeighbors functionality. Rather than choosing a particular nearest neighbors algorithm, we let scikit-learn decide which nearest neighbors algorithm to use after analysis of the training model. Algorithms available include brute force, the K-D tree, and Ball tree. The latter two trade precision off for efficiency by using heuristics to prune which points are considered when evaluating “nearest neighbors.” The decision on which algorithm to use is based on the number of samples, the structure of the data (e.g. sparsity, intrinsic dimensionality), the number of neighbors requested for a query point, and the number of query points. The baseline library ended up choosing the K-D tree algorithm because of the k values we cross validated over, and the one we ended up using for our final model, being less than N/2.

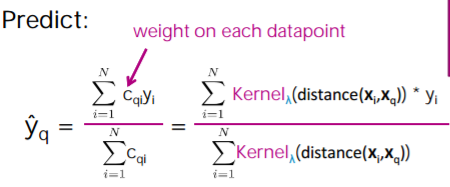
We ran cross-validation over a small subset of our training set (1/10th of the training set). Running cross validation over a set of k values from 1 to 625, we found that a k of 1 minimized our validation error. ****

After running cross validation on our baseline, we implemented our own K-Nearest Neighbors algorithm. We used 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 found the classification with the lowest average Euclidean distance from the query point. That classification was the classification we’d assigned to the query point.

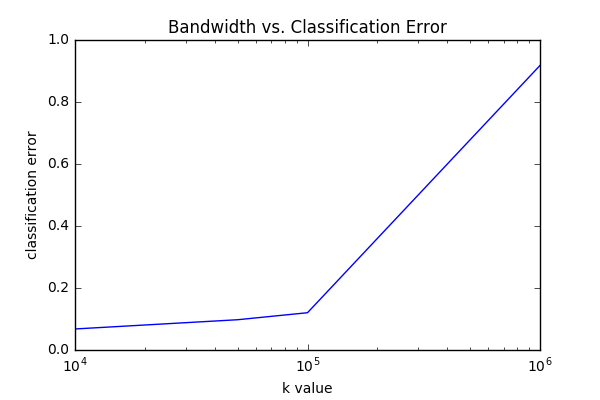


We ran cross-validation over a small subset of our training data (once again, 1/10th). The results backed up the results from the library implementation with a k of 1 minimizing our validation error.

Afterwards, 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.

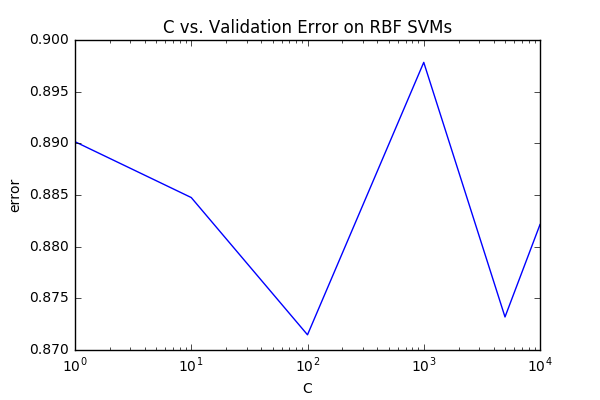
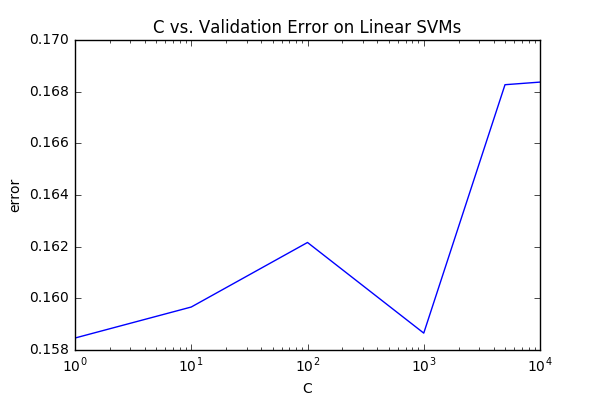
 with 

Since we didn’t 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 104 to 106.



Running cross validation on 1/10th of our training set, we found that a bandwidth of 104 not only avoided overflow errors, it also minimized validation error.

3.2 Support Vector Machines

We then tackled Support Vector Machines. For baselining, we used scikit-learns LinearSVC and SVC functionality. The former trains a Support Vector Machine until convergence with a Linear kernel, and the latter with a Radial Basis function kernel. 

Running cross-validation for each C value over a small subset of the training set (10000 rows), we found that C values of 1 and 100 minimized validation error in the Linear and RBF SVMs respectively. This was surprising to us, especially the former C-value, and we suspected after some tinkering around that the differences in validation error can be ascribed to differences in the validation blocks’ data rather than differences in C value. Nonetheless, we used those respective C-values for creating our model against the the training data and running it against the test data.

3.3 Convolutional Neural Networks

**4 Test Results**

For our final tests, we applied the previously tuned (via cross validation) optimal hyperparameters to their respective algorithms and built models for each of our algorithms. Although we calculated test error off of the entire 10000 point test set, we built each of our models by training against a 25% random sample of our training set (15000 rows). This was a necessity because although some of the algorithms could finish efficiently (e.g. the convolutional neural network), some would take a long time to finish building a model against a 60000 row training set. We predict that with a larger training set for our model to train against, our eventual test error would’ve been smaller. However, for the sake of completing this project on time, that was a trade off we made.

Our test errors ended up being:

|  |  |
| --- | --- |
| **Algorithm** | **Test Error** |
| K-Nearest Neighbors (our implementation, k = 1) | 4.5% |
| Kernelized Nearest Neighbors Regression (Nadaraya-Watson Weighted Gaussian Kernel) | 4.7% |
| K-Nearest Neighbors (SciKit baseline, k = 1) | 4.7% |
| Stochastic Gradient Descent |  |
| PEGASOS |  |
| PEGASOS + Random Fourier Features |  |
| Linear SVM (SciKit) | 88.7% |
| Radial Basis Function SVM (SciKit) | 12.8% |
| Convolutional Neural Network |  |

**5 Conclusions**