An Investigation of Classification Techniques For Handwritten Digit Recognition

Deric Pang
dericp@cs.washington.edu

Saidutt Nimmagadda
nimmas@cs.washington.edu

1 Project Description

1.1 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.

1.2 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.

2 First Considerations

2.1 Machine Learning Techniques

Knowing that we were faced with a classification problem, we immediately considered a multitude of machine learning tecniques such as logistic regression, k-nearest neighbors, perceptron, support vector machine, k-means, and neural networks. In order to narrow our focus to a few methods, we needed to explore existing literature in the domain of handwritten digit recognition.

2.2 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.

3 Approach

3.1 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.

3.2 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.

3.3 k-Nearest Neighbors (k-NN)

3.3.1 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.

3.3.2 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 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.

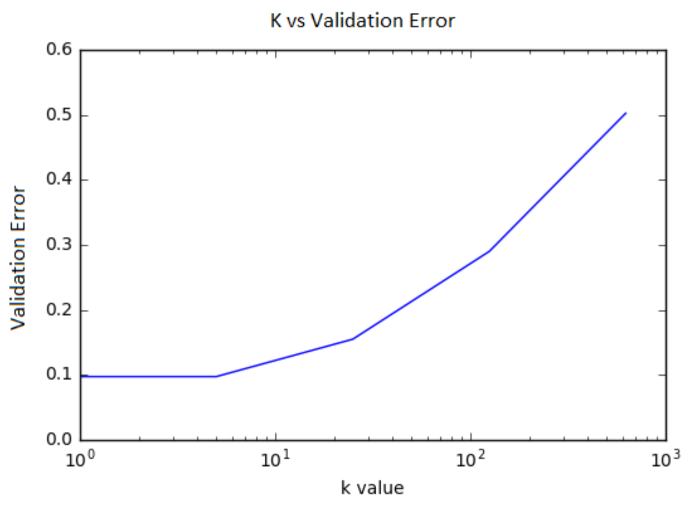


Figure 1: cross validation of scikit-learn's k-NN over varying k values

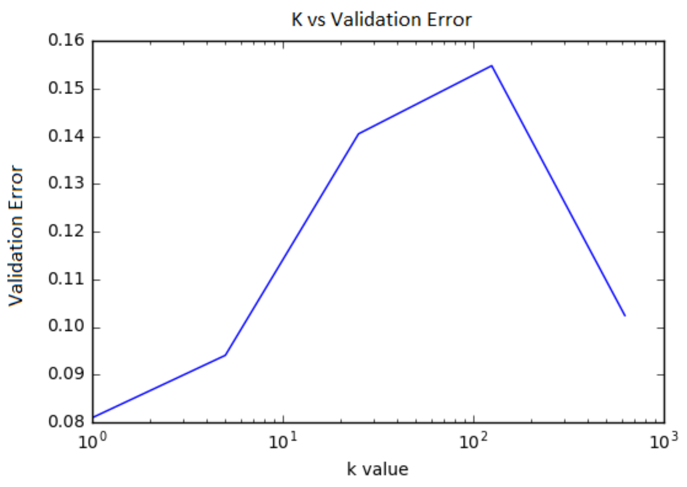


Figure 2: cross validation of k-NN over varying k values

3.3.3 Kernelized k-NN

We have implemented a Gaussian kernel and classified query points based on the Nadaraya Watson Kernel Weighted Average. Instead of finding the k nearest neighbors to classify a query point, we use this kernelized regression to calculate the classification of each query point.

Predict:

$$\hat{y}_q = \frac{\sum_{i=1}^{N} \overset{\text{weight on each datapoint}}{c_{qi}} y_i}{\sum_{i=1}^{N} c_{qi}} = \frac{\sum_{i=1}^{N} \text{Kernel}_{\lambda}(\text{distance}(\mathbf{x}_i, \mathbf{x}_q)) * y_i}{\sum_{i=1}^{N} \text{Kernel}_{\lambda}(\text{distance}(\mathbf{x}_i, \mathbf{x}_q))}$$

Gaussian kernel:

$$\text{Kernel}_{\lambda}(|x_i - x_q|) = \exp(-(x_i - x_q)^2/\lambda)$$

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.

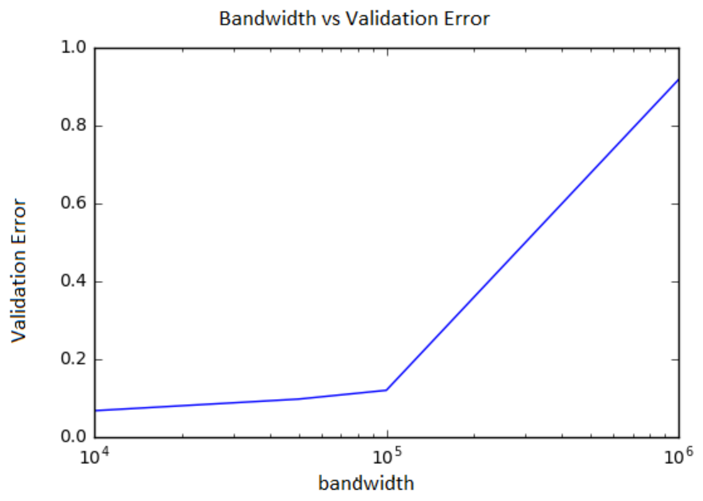


Figure 3: cross validation of kernelized k-NN over varying bandwidth values

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

3.4 Support Vector Machine (SVM)

3.4.1 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.

We noticed that scikit-learn's kernelized SVM performed almost as badly as random guessing, which was very surprising to us. We knew that an RBF kernel was shift invariant and therefore should not be heavily impacted by the fact that the MNIST pixel data is not centered around 0. We concluded that other properties of the MNIST dataset, such as its sparseness, must not lend itself well to kernelization.

3.4.2 SVM Trained With Stochastic Gradient Descent

We have implemented a vanilla SVM trained with SGD. We were able to train this SVM over the entire training set. After 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.

This SVM was fast to train and performed quite well as we will see in section 4.

3.4.3 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. The algorithm can be seen in figure 7:

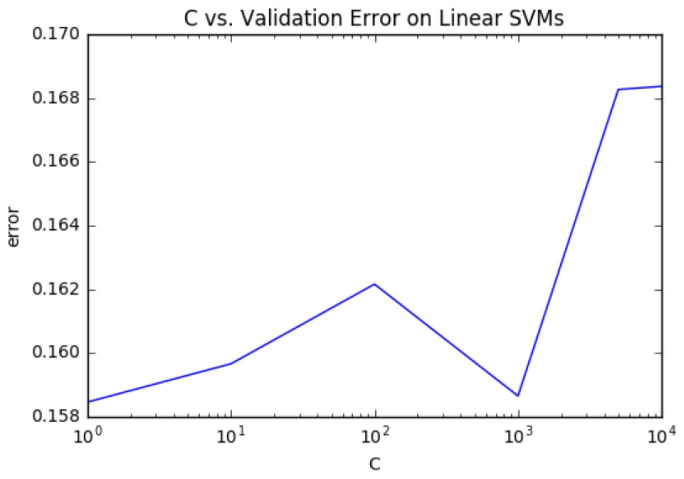


Figure 4: cross validation of scikit-learn's linear SVM over varying C values

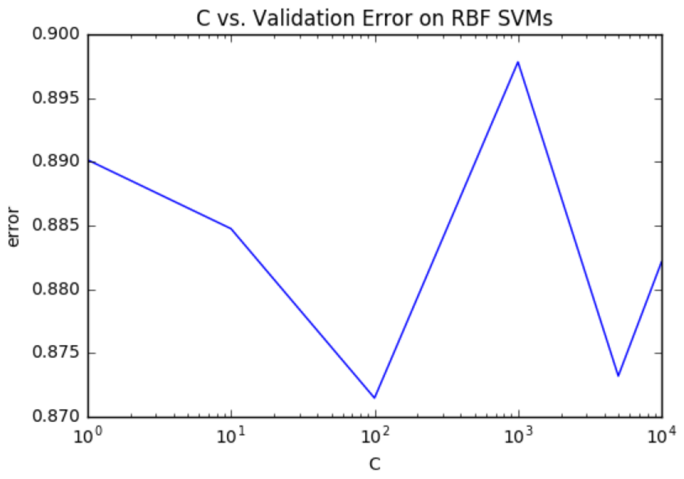


Figure 5: cross validation of scikit-learn's kernelized SVM over varying C values

After 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 8.

Because there is only one hyper-parameter to tune in the PEGASOS algorithm, this SVM was simpler to train. Additionally, it performed slightly better than the SVM trained with SGD.

3.4.4 RBF Kernel Approximation with Random Fourier Features

When we saw how slowly scikit-learn's kernelized SVM trained, we decided to implement a kernel approximation method. We have implemented Random Fourier Features to approxmiate an RBF kernel [5]. The algorithm can be found in figure **??**.

We confirmed that our implementation of Random Fourier Features was correct by applying it to a different domain (large scale classification in map-reduce [9]) and confirming that the RBF kernel approximation indeed reduces classification error. However, much like scikit-learn's RBF Kernel SVM, our RFF SVM performed quite poorly—nearly as bad as random guessing. We concluded, like before, that there must be some aspect of the MNIST dataset that does not lend itself well to kernelization.

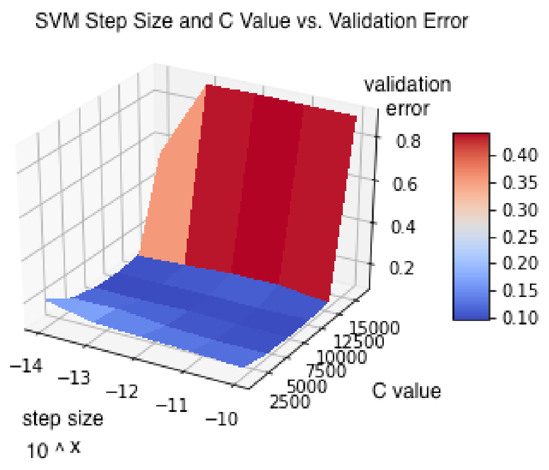
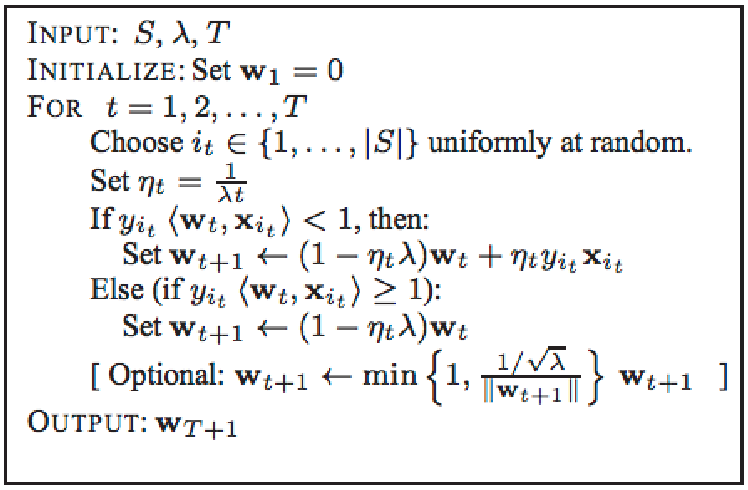


Figure 6: cross validation of SGD SVM over varying step sizes and C values



```
INPUT: $S, \lambda, T$
INITIALIZE: Set $\mathbf{w}_1 = 0$
FOR $t = 1, 2, \ldots, T$
    Choose $i_t \in \{1, \ldots, |S|\}$ uniformly at random.
    Set $\eta_t = \frac{1}{\lambda t}$
    If $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1$, then:
        Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \eta_t y_{i_t} \mathbf{x}_{i_t}$
    Else (if $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle \geq 1$):
        Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t$
    [ Optional: $\mathbf{w}_{t+1} \leftarrow \min \left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\} \mathbf{w}_{t+1}$ ]
OUTPUT: $\mathbf{w}_{T+1}$
```

Figure 7: PEGASOS algorithm

3.5 CNN

In order to achieve near state-of-the-art performance, we also implemented a convolutional neural network using TensorFlow [7]. Our neural network has two convolutional layers, one densely connected layer, and uses rectified linear units as well as dropout. This neural network was quite simple to implement in TensorFlow, and it performed significantly better than all of the previous methods.

Most hyper-parameters, like the learning rate of the Adam [8], have been suggested by TensorFlow. This meant that the only hyper-parameter we had to tune was the number of training steps to take. We found that at around 20,000 training steps, the neural network performed extremely well.

4 Results

For our final tests, we applied the optimal hyper-parameters found through cross-validation to their respective algorithms and built models for each of our algorithms. We calculate test error off of the same MNIST test set. Each model was trained as described in section 3. These test errors are presented in figure 10.

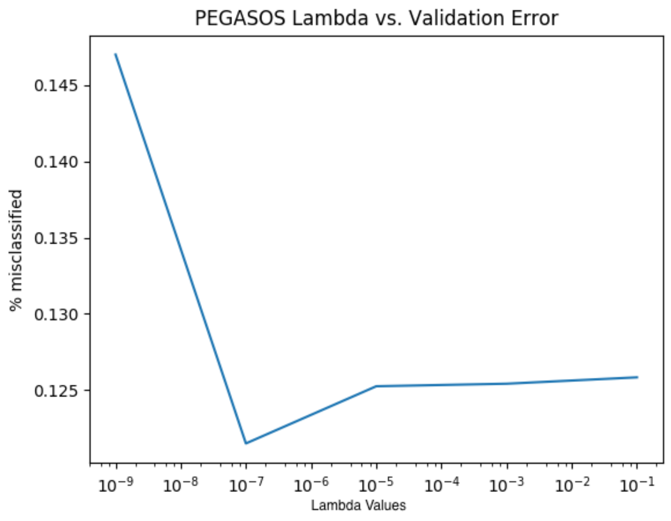


Figure 8: cross validation of PEGASOS SVM over varying λ values

```
Algorithm 1 Random Fourier Features.
Require: A positive definite shift-invariant kernel $k(\mathbf{x},\mathbf{y}) = k(\mathbf{x}-\mathbf{y})$.
Ensure: A randomized feature map $\mathbf{z}(\mathbf{x}) : \mathcal{R}^d \to \mathcal{R}^D$ so that $\mathbf{z}(\mathbf{x})'\mathbf{z}(\mathbf{y}) \approx k(\mathbf{x}-\mathbf{y})$.
  Compute the Fourier transform $p$ of the kernel $k$: $p(\omega) = \frac{1}{2\pi}\int e^{-j\omega'\delta}k(\delta)\, d\Delta$.
  Draw $D$ iid samples $\omega_1, \cdots, \omega_D \in \mathcal{R}^d$ from $p$ and $D$ iid samples $b_1, \ldots, b_D \in \mathcal{R}$ from the uniform distribution on $[0, 2\pi]$.
  Let $\mathbf{z}(\mathbf{x}) \equiv \sqrt{\frac{2}{D}}\left[\cos(\omega_1'\mathbf{x}+b_1) \cdots \cos(\omega_D'\mathbf{x}+b_D)\right]'$.
```

Figure 9: Random Fourier Features

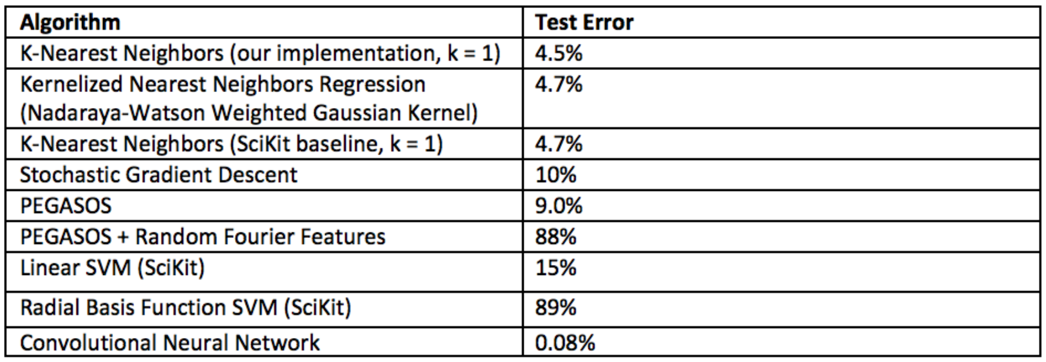
5 Conclusions

We were suprised by how effective the nearest neighbor algorithm was. Our implementation of k-NN performed similarly to the scikit-learn baseline with a 4.5% test error compared to the baseline's 4.7% test error. We ascribe the negligible disparity in test error due to a difference in which points were sampled from the training set and constituted our model. The kernelized regression disappointed us with its 4.7% test error, which was higher than the unkernelized k-NN regression. However, we suspect that if it weren't for overflow issues that restricted us to bandwidths greater than 10^4, we could've tuned a bandwidth that made kernelized regression have an even lower test error than unkernelized k-NN. For both algorithms, running on the entire dataset rather than a small fraction would likely reduce test error, but the runtime would simply be intractable.

Our SVM impressed us. We implemented stochastic gradient descent over a set number of epochs, and we beat the performance of the library SVMs that ran to convergence. With the PEGASOS algorithm, we saw a slight increase in accuracy. What surprised us was how the introduction of Random Fourier Features to approximate a Radial Basis Function kernel was not significantly better than random guessing. The Radial Basis Function kernel's inadequacy in the baseline scikit-learn SVM also suggests that properties of the data itself rendered the radial basis kernel and random fourier features techniques ineffective in classification.

In terms of accuracy, the convolutional neural network was unparalled. The CNN was quick to train, and performed spectacularly.

In conclusion, it is clear to see the advantages of the CNN. Given a large dataset, it has an extremely high accuracy. That being said, the major takeaway was an appreciation for the effectiveness of



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	10%
PEGASOS	9.0%
PEGASOS + Random Fourier Features	88%
Linear SVM (SciKit)	15%
Radial Basis Function SVM (SciKit)	89%
Convolutional Neural Network	0.08%

Figure 10: final test errors of the different classifiers

simple techniques. Kernelization is not guaranteed to improve accuracy and can be difficult to implement corectly for a dataset. We were also extremely impressed with how accurate classification can be when all you do is take a guess based off of the closest point. Given what we had read in the related work, we did not expect such a result.

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.

[7] Abadi, Martn, et al. ”Tensorflow: Large-scale machine learning on heterogeneous distributed systems.” arXiv preprint arXiv:1603.04467 (2016).

[8] Kingma, Diederik, and Jimmy Ba. ”Adam: A method for stochastic optimization.” arXiv preprint arXiv:1412.6980 (2014).

[9] https://github.com/dericp/large-scale-classification