# CS 440: MDPs, Reinforcement Learning, and Classification

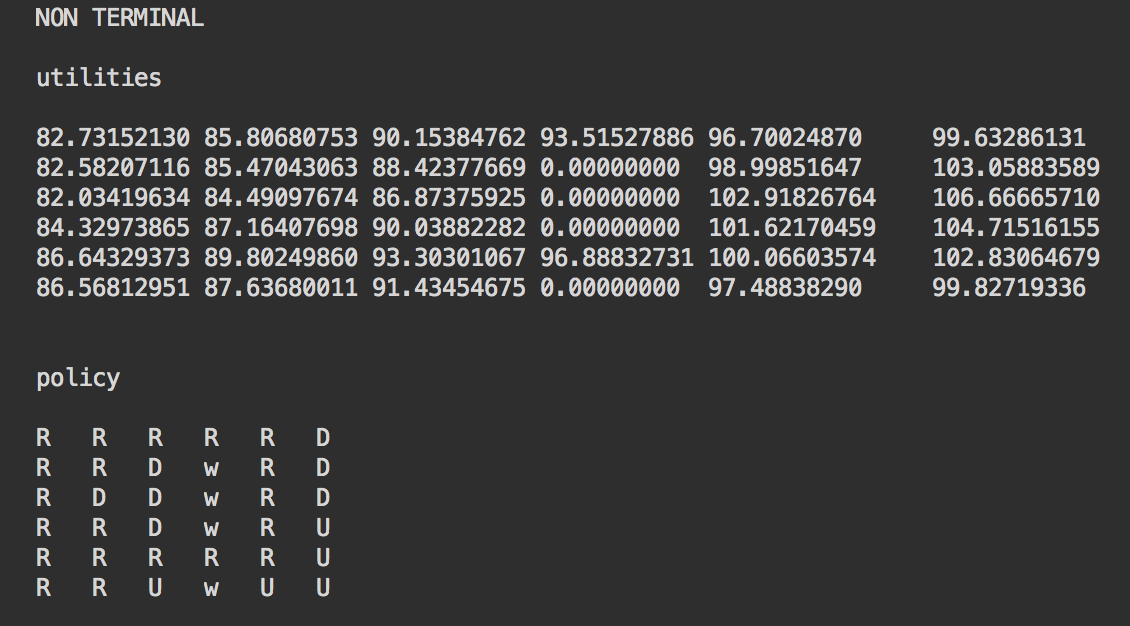
## Part 1: Grid World

### Grid World MDP

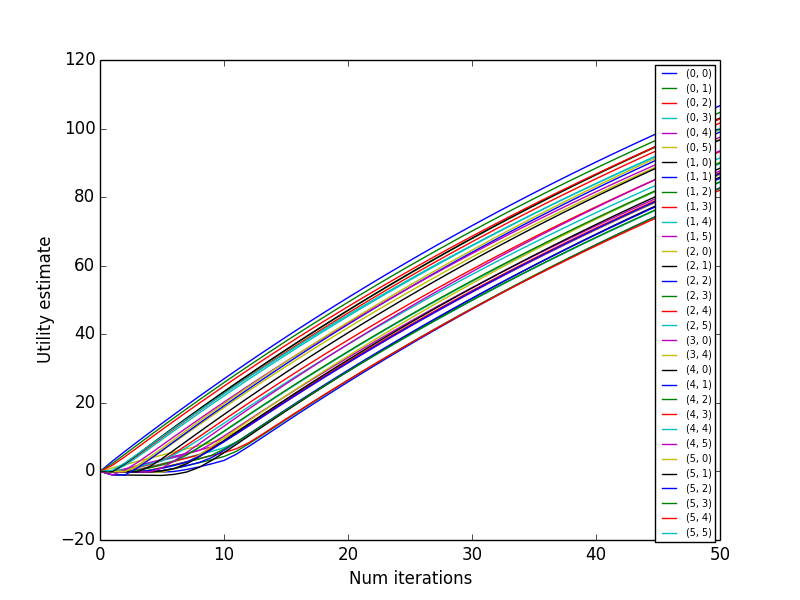
We implemented Value iteration, as described in the lecture slides. For each state s, the Utility would be updated as shown Utility(s) = Reward(s) + (Gamma \* E(s)). E(s) was a function that figures out the optimal move from a given state s and returns the utility of that move. The optimal move is determined by taking (.8\*the intended move + .1\*right + .1\*left) for each possible move from the given state. Right and left in the above equation indicate the two moves perpendicular to the intended move. Then, the move with the highest number is deemed the optimal move. It is an expected utility function. Gamma was our discount factor, which was .99. Overall, we did fifty iterations of updating the utility of each state.

### Non-Terminal Reward Squares

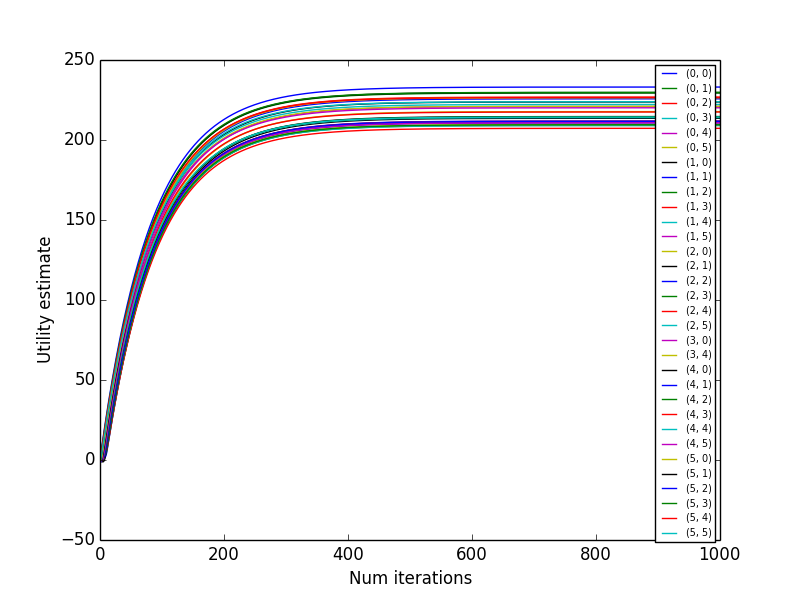
‘w’ in the figure below represents a wall. The utility of a wall is labelled 0. Utilities are truncated to 8 decimal places.



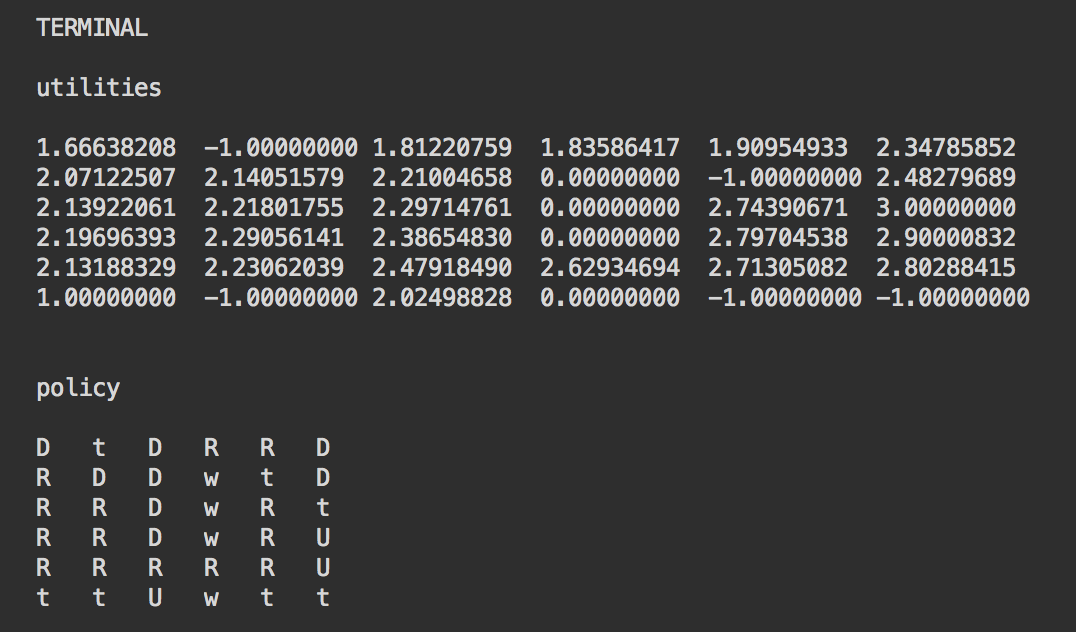
Below is our plot for non terminal states.



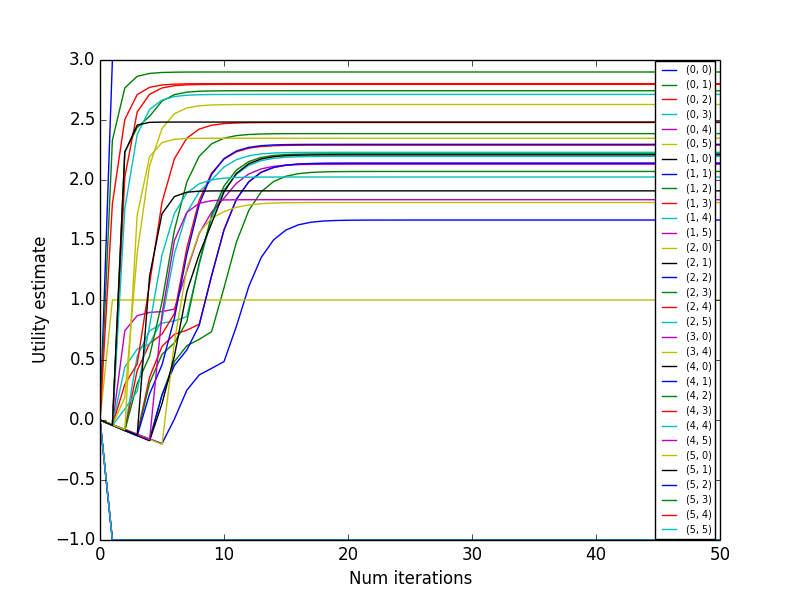
Since the non-terminal seems to need more iterations, here is a larger plot.



### Terminal Reward Squares

‘w’ in the figure below represents a wall. ‘t’ in the figure below represents a terminal state. The utility of a wall is labelled 0. Utilities are truncated to 8 decimal places.

Below is our plot for terminal states. The utility of every state starts at 0. For terminal states, the utility becomes the reward of that state after the first iteration.



### Grid World Reinforcement Learning

We implemented TD Q-Learning as described in the lecture slides.

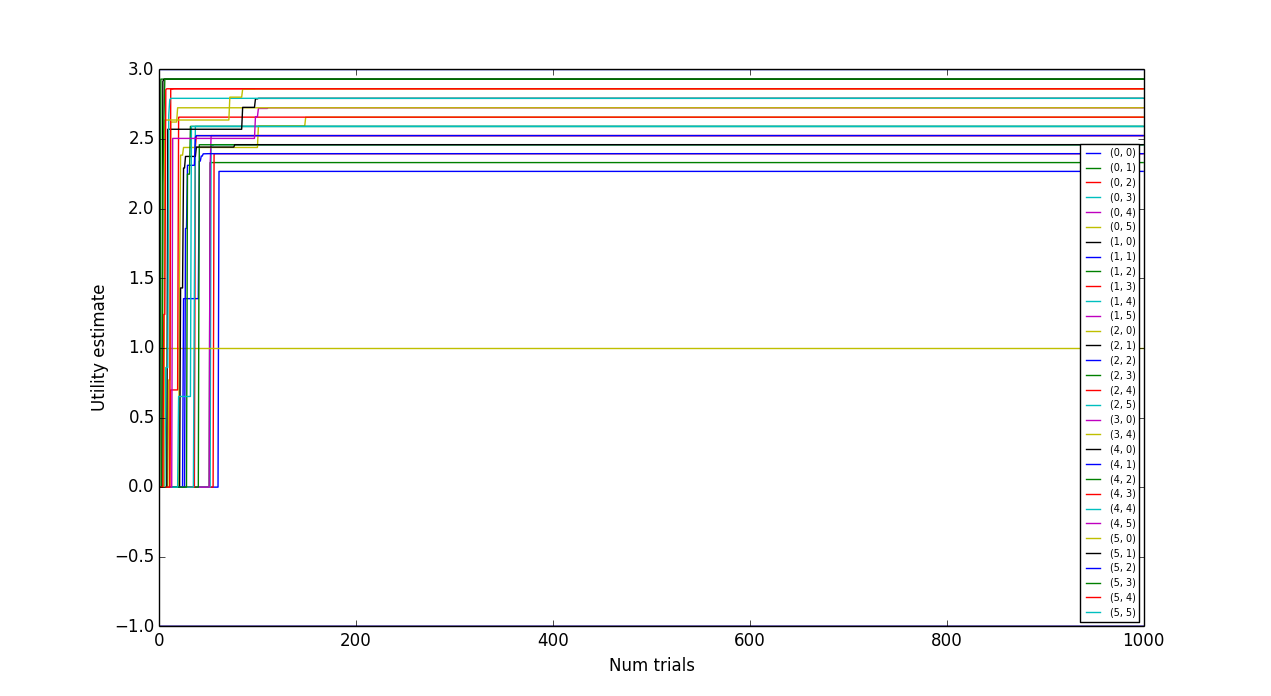
For our learning rate function, we started with alpha(t) = 60/(59+t), and we iterated from 1 to 50 as our values of t. For a given state, we iterated over each value of t. At each iteration we determined the optimal action using our exploration function, and then updated the value of Q(state, action) using the formula displayed in the lecture slides. We also incremented the number of times the given state and action combination was executed for future use in our exploration function.

We experimented with our exploration function. We settled on using an exploration function similar to the one in the lecture slides. Here is a layout of our exploration function. ‘u’ is Q(state,action) and n is the number of times a given state and action combination has been executed.

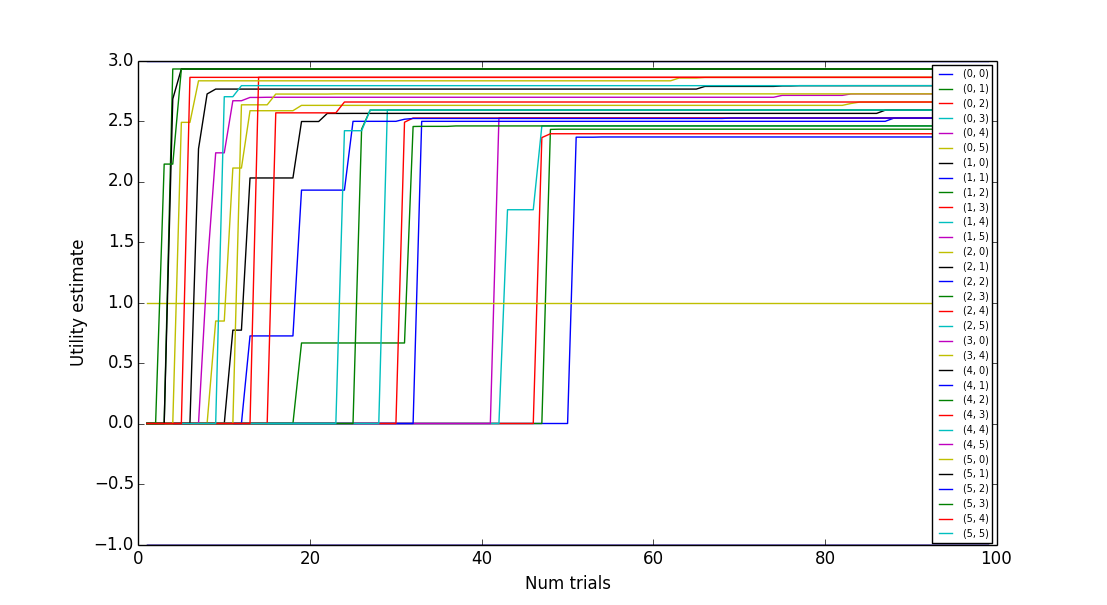
E(u,n): if( n < Ne) return MAX\_INT; return u;

The parameter tweaking we did was in deciding a value for Ne. We experimented with values ranging from 1 – 100, and we chose 20 in the end. Values near 20 seemed to yield the best results, including the low RMS error as seen below.

Below is a plot for utility estimates plot.

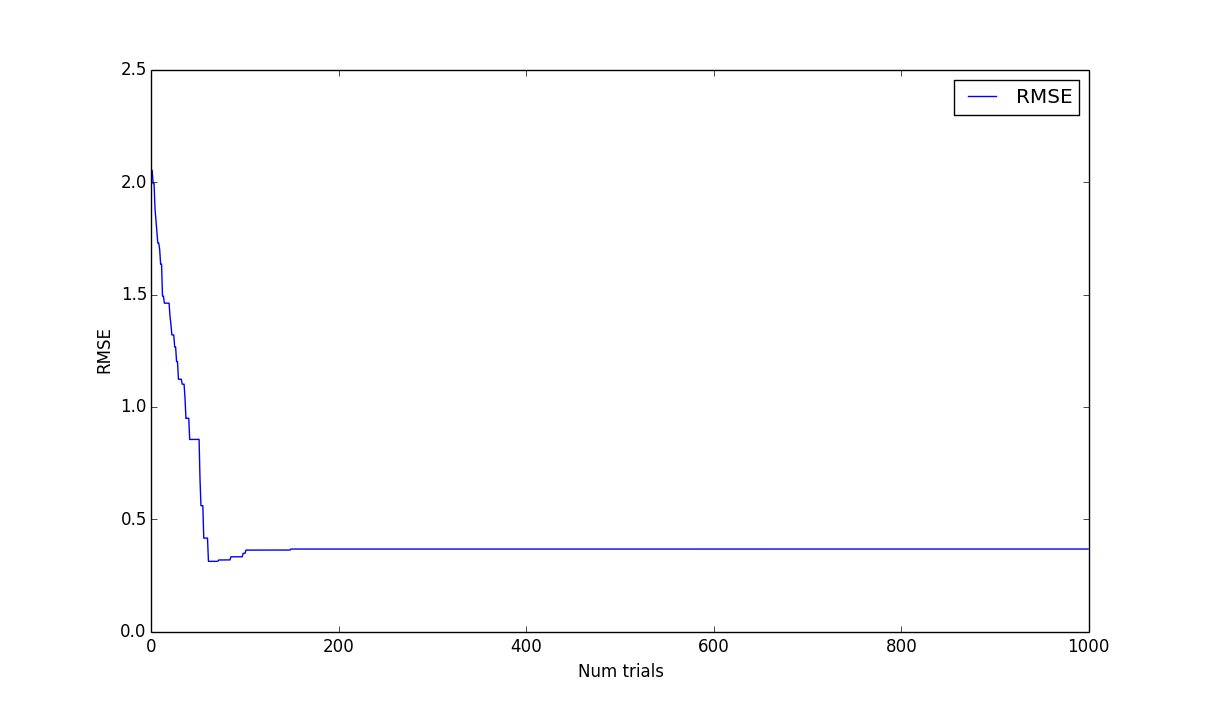


A more compact version of that same plot is shown below.



The reason both of these plots do not look smooth is because a state is chosen a random number of times. You do not necessarily look at every state the same number of times. Though the number of times that a state is chosen is randomized, we ran our code for sufficiently many iterations to ensure accuracy.

Below is our RMS error plot.



## Part 2. Digit Classification

### 2.1 Digit Classification with Perceptrons

We vectorize each image. Essentially, each image becomes a vector of length 784, with each element of the vector being a feature. The image is decomposed so that blank pixels are given the value 0 and non-empty pixels are given the value 1. (We did attempt to differentiate between grey and black pixels, however this negatively impacted our overall accuracy, see below).

We generate two arrays. The first is an array that holds the labels, and second is an array that holds the vectorized images. Indices are corresponding across arrays.

Our classifier uses a stochastic gradient descent method to train the perceptrons. The training set of images X, and the set of training labels y were passed into the perceptron. We train the data across 50 epochs (see below), each epoch performs 100 gradient descents to update the weight vector for that particular category. The classifier itself updates the weight vector according to what the current prediction for the current weight vector and bias is. The weight vector and bias are updated according to the gradient of the cost. An incorrect classification updates the weight vector in the direction of the other category, while a correct classification updates the weight vector in the direction of the “support vectors” in order to prevent overfitting.

If an image is correctly identified, then the bias of that label does not change. If an image is incorrectly identified, the bias is -1 \* the correct label of the image. If an image is correctly identified, the weight of that label changes very slightly. We do this to prevent overfitting of the data. If an image is incorrectly identified, the weight of the predicted label (not the correct one) is updated according to the direction of the gradient’s descent.

Our initial implementation performed well (~79% accuracy), however every category had its own perceptron that was trained individually. This led to our runtime being in the order of ~150 seconds. A better implementation increased our accuracy to ~82%. This better implementation made a single “classifier” instance that kept track of the weight vectors for each category. The training set passed in was used once - the classifier simply ran its STG weight update across all categories. This led to a tenfold decrease in runtime along with a ~3-4% increase in accuracy: with testing across the entire training set the runtime hovered at ~15 seconds.

We experimented with all of the following categories.

#### Learning Rate Decay Function

We settled on using 1/(1+epoch). We also tried other learning rate decay functions, including 1000/(1000+epoch) and 1/(2+epoch), 2/(2+epoch) and a few more.

#### Bias vs. No Bias

We settled on using a bias that is initialized to zero. This worked better than having a random initialization of bias, with values being between -1 and 1. This also worked better than having no bias at all.

#### Number of Epochs

We settled on using 50 epochs. We experimented with values up to 100, and deemed that 50 was best.

#### Initialization of Weights (zeros vs random)

We settled on using zeros. This works better than a random initialization of weights

#### Ordering of Training Examples (fixed vs. random)

We settled on using random ordering of training values rather than a fixed ordering of training values.

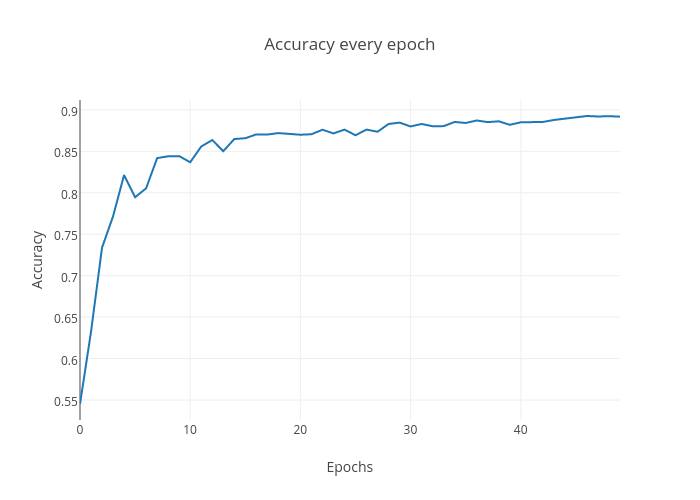
#### Binary and Ternary Feature Inclusion

We experimented with treating gray values differently, but eventually settled on using purely binary features. Whitespace would be represented as a white color pixel, and all non-whitespace characters would be represented as black color pixels. Surprisingly, using gray features negatively impacted our accuracy. This was for bonus points.

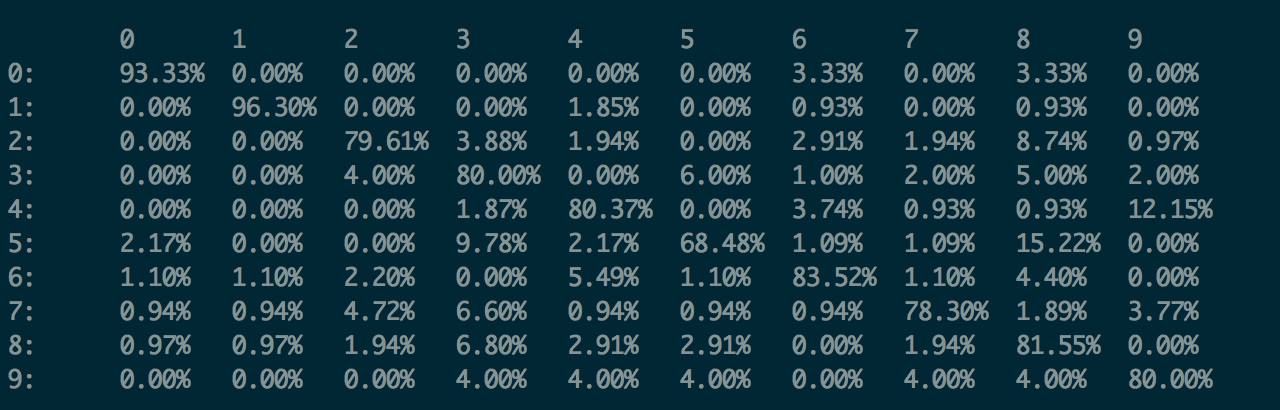
#### Regularization Constant

We settled on using a regularization constant of 0.001. We experimented with values up to 1, and deemed 0.001 best. This was for bonus points.

Our training curve is shown below. Naturally, your accuracy is high at larger epochs, as your weights have been consistently modified.



Here is our confusion matrix.



Our overall accuracy was 82.0%, in comparison to 77.1% from Naïve Bayes in Assignment 3.

For bonus points, we implemented a stochastic gradient descent model. In addition, we used a regularization constant. We also experimented with bias values that were beyond just 0 and 1. The bias model that we used consistently changes to minimize errors.

### 2.2 Digit Classification with Nearest Neighbor (Bonus Points)

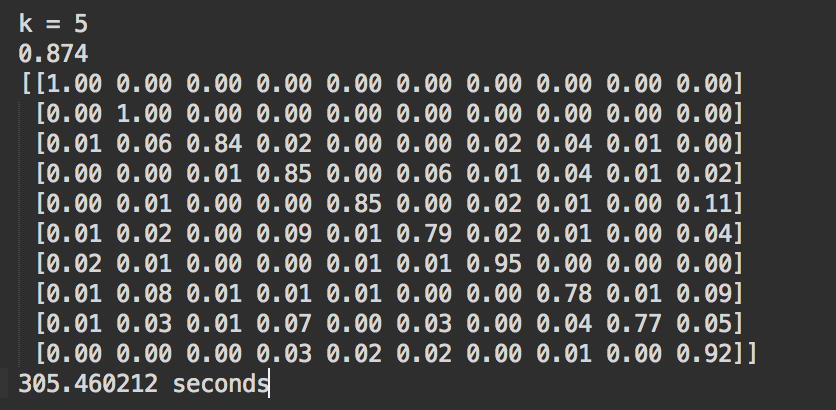
As we are three-unit students, this is for extra credit.

Our distance function takes in two images. Then, for each pixel in the image, it calculates the difference between the pixel values and squares that difference. It then sums those values for all pixels and takes the square root. We experimented with other distance functions, including some of the scipy distance functions, including Euclidean and Manhattan.

Accuracy as a function of k. The validation set was a 25% of the training data. The subset used for the validation set was randomly chosen. This was done to prevent overfitting as well as to increase runtime.

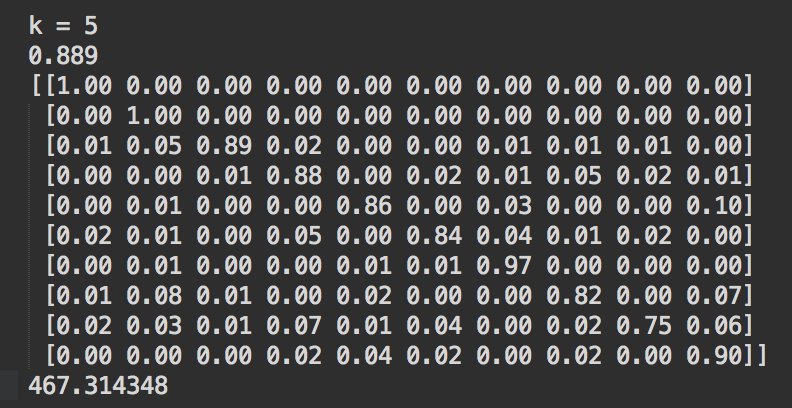
|  |  |  |
| --- | --- | --- |
| k | Accuracy | Time (s) |
| 1 | 0.836 | 155.28 |
| 2 | 0.812 | 152.41 |
| 3 | 0.846 | 157.91 |
| 4 | 0.817 | 156.97 |
| 5 | 0.849 | 153.48 |
| 6 | 0.834 | 153.62 |
| 7 | 0.847 | 153.81 |
| 8 | 0.831 | 153.48 |
| 9 | 0.828 | 153.62 |

For k=5, with a validation set of 50%, results are shown below. Our overall accuracy was 87.4% with k as 5. Runtime was relatively similar between all values of k, as shown above.



|  |  |
| --- | --- |
| Nearest Neighbor | 87.4% |
| Naïve Bayes | 82.0% |
| Perceptron | 77.1% |

With a validation set of 75%, here are some results. It appears that there is a marginal improvement in accuracy, but a very heavy time cost to increasing the size of the validation set.

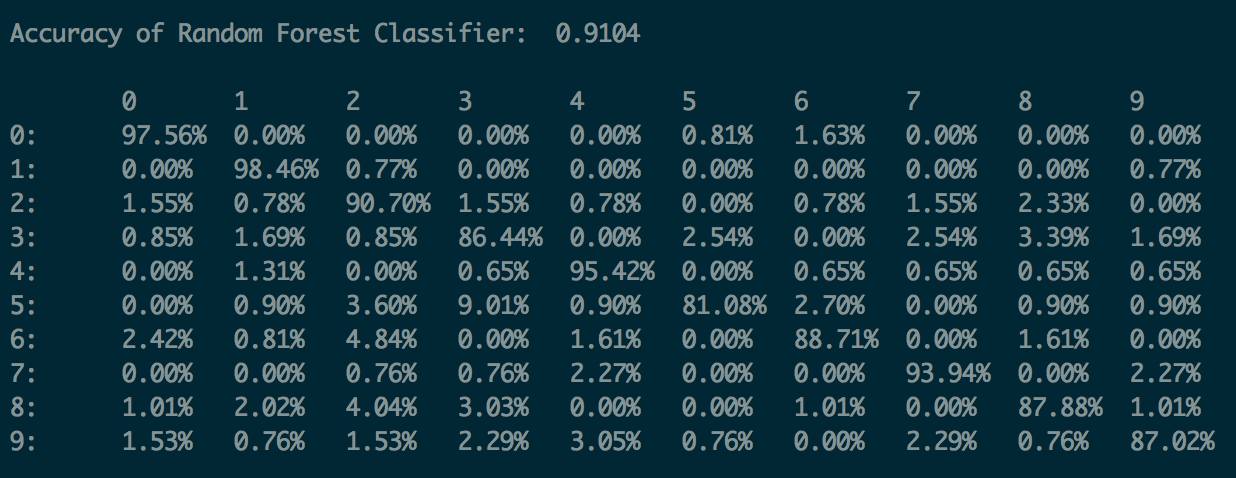


### 2.3 Digit Classification with Random Forest Classifier (Bonus Points)

This work is for extra credit.

We used the Random Forest Classifier from the SciKit Learn library on our digit dataset. Here is a link to the library. <http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

Here are the results of that classifier.



## Individual Contribution

### Abhishek Nigam

We both worked together for the entirety of the project. However, the bulk of my efforts were spent on Part 1. I also did Nearest Neighbor Classification.

### Jakub Klapacz

We both worked together for the entirety of the project. However, the bulk of my efforts were spent on Part 2. I also did the Random Forest Classification.