# CS391L Machine Learning: HW1 Eigendigits

Joel Iventosch

February 7, 2015

## 1 Introduction

The goal of this assignment was to classify images of hand written digits using eigenvector decomposition of covariance for analysis - also known as principle components analysis ( PCA). While this assignment involves a specific application of PCA to the domain of hand-written digits, from a higher-level perspective this assignment was intended to teach us about PCA in a more general sense. It was very effective in demonstrating how PCA can effectively be applied as a data pre-processing tool in classification problems. Although the dimensionality of this particular domain was relatively low-scale (e.g. 28x28 pixel images are a fraction of the dimensionality of typical higher-resolution images), one can easily see how these same techniqes could be applied to larger real-world problems. Throughout the process of crafting my code, the power of PCA was revealed - namely, that it can be used to effectively solve problems in very large-scale, high-dimensionality domains where the curse of dimensionality would make the problem intractible (or at least very expensive) using typical classification methods with out the assistance of PCA. By using PCA to pre-process the data and reduce its dimensionality, these costly problems become much more managable. The true power and beauty of PCA also lies in the fact that it not only reduces the dimensionality of large-scale domains, but in many cases it actually improves the ability of the underlying model to perform its task (e.g. classification), by "cleaning" the data in the process of dimensionality reduction and removing noise that might cause confusion otherwise. Thus the power of PCA as a pre-processing mechanism is two-fold: reduce the cost and complexity of a problem - AND improve the performance of the associated model.

## 2 Methodology

PCA works by creating an orthogonal mapping of a matrix from its original dimensionality into a reduced dimensionality. This is implemented by taking the eigenvectors of the covariance matrix of the original data set matrix. Althoug this new eigen vector matrix initially still has the same dimensionality of the original covariance matrix, it can be reduced by ordering the eigen vectors in the eigen vector matrix according to the magnitude of their correspoding eigen values (from left to right), and then only keeping the top $k$ eigen vectors. This results in an eigen vector matrix with the same number of rows as the original covariance matrix, but with far fewer columns.

For this homework we used the above process in addition to another "trick" to help reduce the dimensionality of the domain. We started with a training data set matrix of order $784x60,000$. I then selected the first k columns of that data set to form my training matrix, $A$. Thus $A$ is a $784xk$ dimensional matrix. The core mathematical function in my program (hw1FindEigendigits) then finds the mean column vector $m$ of $A$ and then redefines A by subtracting $m$ from each column of $A$. I then find $A^T$, at which point I use the "trick" dicussed in class to find the eigenvectors of $A$. I implement this trick by left multiplying $A$ by $A^T$, which results in a "covariance-like" matrix $A^TA$ of dimensionality $kxk$ (hypothetically a much smaller matrix than $784x784$, although not necessarily the case in this domain). Next, I find the eigen vectors of $A^TA$ which results in a $kxk$ matrix of the eigen vectors of our "covariance-like" ma-

trix $A^TA$. What the trick learned in class reveals, however, is that by mulitplying each of the eigen vectors, $v$ of $A^TA$ by $A$, namely, $Av$ for each $v$ in the eigen vector matrix of $A^TA$ we get the eigen vectors of our ***actual*** covariance matrix, $AA^T$ (which is of dimensionality $784x784$). By ordering these new eigen vectors (which are now each $784x1$ column vectors) from left to right according to the magnitude of their corresponding eigen values, the result is a matrix, $V$, of dimensionality $784xk$, which contains the eigen vectors of our covariance matrix $AA^T$ and acts as a "mapping" of our data from the full-dimensional space into the eigenspace. Although this mapping does not yet constitute a dimensionality reduction, we can peel off the eigen vectors that correspond to the lower-valued eigen values, keeping only the top set of eigen vectors, thus reducing the size of $V$ from $k$ columns to far fewer. This reduced eigen vector matrix $V$ now maps each training and/or test vector into an eigenspace of reduced dimensionality.

I implemented the dimensionality reduction aspect of this process in a slighlty different way is typically done (and perhaps differently than I would have done if doing the process again). Instead of finding $V$ and then keeping only the top set of those eigen vectors, I used all $k$ eigen vectors in $V$, (let's call this $k_{mapping}$) but then used a much larger $k$ value to create my $784xk$ matrix of training data in the acutal classification process (we'll call this $k_{training}$). Although this is a different route to dimensionality reduction, the result is similar. The one aspect of the process that my methodology misses on, is that it doesn't necessarily optimize the set of eigen vectors being used in $V$ as well as it could have. In other words, I use the first $k_{mapping}$ training column vectors of the training data matrix ($784x60,000$) to create my mapping matrix, $V$ and then I use the next $k_{training}$ column vectors of the trianing data matrix to create a pool of pre-labeled targets for use in comparison in nearest neighbor classification...where $k_{mapping} < k_{training}$ (at least for most experiments). The main reason for doing this was that it didn't seem to make sense to have $k_{mapping} > 784$ since the result would have actually increased the dimensionality of the problem because the dimensions of the "covariance-like" matrix $A^TA$ would then be greater than $784x784$, thus negating any value the trick we learned in class might have added. I recognize however, that in larger-scale real world problems (where our $28x28 = 784$ dimension would likey be more on the order of $256x256 = 65,536$ pixels) one could select a much higher number for $k_{mapping}$ and still acheive dimensionality reduction in the calculation of the eigen vectors (by using the "covariance-like" matrix). That said, although my methodology was slightly different and perhaps not as optimal, it still produced quality results with a top accuarcy of 98 percent.

To conclude this section on methodology I will breifly describe the overall process I used:

1. Select $k_{mapping}$ columns of training data ($k_{mapping} < 784$) to create $A_{initial}$
2. Find the mean column vector $m$ of $A_{initial}$
3. $A = A_{initial} - m$
4. Create $V$ from $A^TA$ as described above
5. Select the next $k_{training}$ column vectors from the training data set to form $A_{training}$
6. Map each column vector in $A_{training}$ into the reduced eigenspace by multiplying by $V^T$
7. Map each test vector, $v$ that I want to test into the eignspace by multiplying by $V^T$
8. Find the norm of the difference of $V^Tv$ and each each vector in $A_{mapping}$ (after vectors of $A_{mapping}$ are projected into reduced eigenspace)
9. Let the 5 nearest neighbors of each test vector vote to determine its projected target label
10. Compare this to its actual label to track the classification accuracy
11. Output results to console and file to track the results

# 3 Experiments

To test the accuracy of my classifier, I ran 63 different experiments (see table and figures below). I had 6 different parameters built into my model that I could potentially vary at runtime (via console input). These are:

1. mapping size (i.e. $k_{mapping}$) from above
2. reuse mapping data in training data used for classification? (i.e. is $k_{mapping}$ be a subset of $k_{training}$ ?)
3. number of training data column vectors to use in training (i.e. comparison) phase of classification ($k_{training}$)
4. number of test data column vectors to test the model on
5. use the easy or hard test data?
6. number of neighbors to use in nearest neighbor classification

Ideally I would have liked to run a Design of Experiments (DOE) to fully test the impact of each parameter on the model. However, given the number of possible parameters and the time and scope of this project, that unforunatley wasn't a realistic option, since that would have required a minimum of $6P6 = 720$ different experiments (and that's assuming each parameter is boolean, which they obviously are not). As such, I chose to vary:

- the number of eigen vectors used in my mapping size, $k_{mapping}$
- the number of training vectors used in my nearest neighbor comparison, $k_{training}$
- and the difficulty level of the test data used

while holding the rest of the parameters constant:

- reuse mapping data in training data used for classification = **NO**
- number of test data column vectors to test the model on = **500**
- number of neighbors to use in nearest neighbor classification = **5**

Note, I did initially run several experiments with larger sets of test data (e.g. 1,000 and 5,000 test images) and the results appeared to be almost identical to those experiments run with 500 test images - thus for the sake of processing speed and time, I chose to stick with 500 test images. The results of these experiments can be visualized in the table and figures at the end of the report and are discussed in the next section.

# 4 Results and Discussion

The results of my experiments were pretty interesting. Overall, I was very pleased with the levels of accuracy that I was able to acheive. My top level accuracy was 98+ percent, which was attained in several different experiments. Most of my experiments resulted in $85-95$ percent accuracy, while some (particularly those using very low values for $k_{mapping}$ and $k_{training}$) resulted in lower accuracy in the $55-80$ percent range. As expected, the number of eigen vectors used ($k_{mapping}$) to create the mapping matrix $V$, had an impact on the accuracy of the model as did the number of training vectors $k_{training}$. However, what was interesting was the manner and ranges within which they affected the accuracy. When going from $k_{mapping} = 10$ to $k_{mapping} = 100$ the change in accuracy was very noticeable. However, adding more eigen vectors to the mapping matrix after 100 resulted in minimal additional improvement to the accuracy - and, in fact, even hurt the accuracy at times (particulary when $k_{mapping} = 750$). Increasing the value of $k_{training}$ led to significant improvements in accuracy as well. However, unlike the case with $k_{mapping}$, adding more and more training vectors to the classification set (i.e. continuing to increase $k_{training}$) kept on improving the results all the way up to $k_{training} = 10000$, with less than expected diminishing returns as $k_{training}$ grew. While this was somewhat surprising at first, upon further review it seems to make sense since the nearest neighbor classification technique is a "location-based" type of classifier. Thus, the more training data the "digit map"

is loaded with, the higher the odds are of correctly classifying the test digits. The $k_{mapping}$ parameter, on the other hand, requires slightly more fine-tuning since it is actually creating the mapping from full-dimensional space into the reduced-dimensionality eigenspace. Since this mapping is based on the correlations of the data, it makes sense that increasing $k_{mapping}$ would help to an extent, but then see diminishing returns and even start to hurt as $k_{mapping}$ gets too big and the mapping is "over tuned". Also, as expected, it is worth noting that the model performed uniformly better on the easy test data set than on the hard test data set.

There are a few things that, time-permitting, I would have liked to have done differently and that are left for future work. Among these are: alter and improve my methodology for selecting $k_{mapping}$ (as discussed in the above section), run experiments in which the number of nearest neighbors varies, and run experiments on larger test data sets and with images of higher dimensionality (e.g. $256x256$ pixels). One last item for future work would also be to craft a genetic algorithm to wrap around this model in order to optimize the various parameters discussed above. All said, I learned a lot from this assignment and, although there is always room for improvement with these types of models, I am fairly happy with the accuracy results I was able to attain.

Plots and tables are shown below:

**Easy Test Data**

| Accuracy | # eigvecs (k_mapping) | # training vecs (k_training) | test vectors | easy/ hard |
|---|---|---|---|---|
| **98.0%** | 250 | 10000 | 500 | easy |
| **98.0%** | 100 | 10000 | 500 | easy |
| **98.0%** | 250 | 10000 | 500 | easy |
| **98.0%** | 750 | 10000 | 500 | easy |
| **97.8%** | 500 | 10000 | 500 | easy |
| **97.5%** | 500 | 20000 | 5000 | easy |
| **97.5%** | 500 | 20000 | 5000 | easy |
| **97.4%** | 100 | 5000 | 500 | easy |
| **97.4%** | 100 | 5000 | 500 | easy |
| **97.4%** | 750 | 5000 | 500 | easy |

| | # top eigen vectors used in mapping | | | |
|---|---|---|---|---|
| k_training | top 100 | top 250 | top 500 | top 750 |
| **100** | 78.0% | 74.4% | 75.0% | 77.8% |
| **500** | 90.4% | 91.6% | 89.0% | 88.4% |
| **1000** | 93.4% | 93.4% | 92.2% | 92.6% |
| **5000** | 97.4% | 97.2% | 97.2% | 97.4% |
| **10000** | 98.0% | 98.0% | 97.8% | 98.0% |

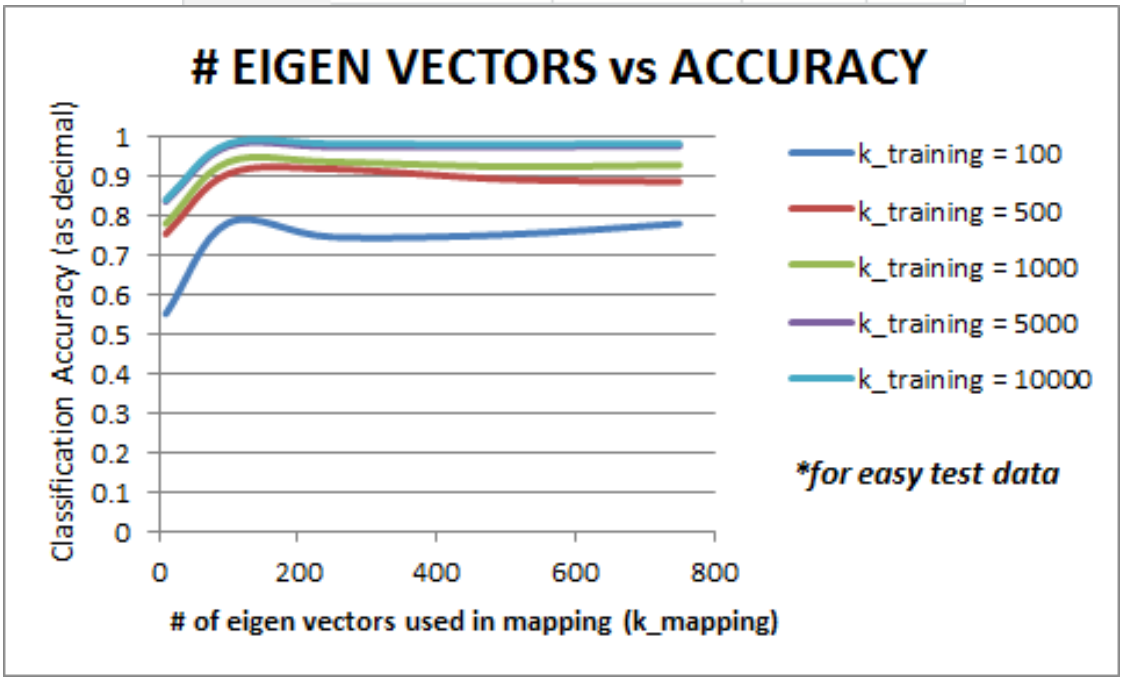
<< Easy Test Data

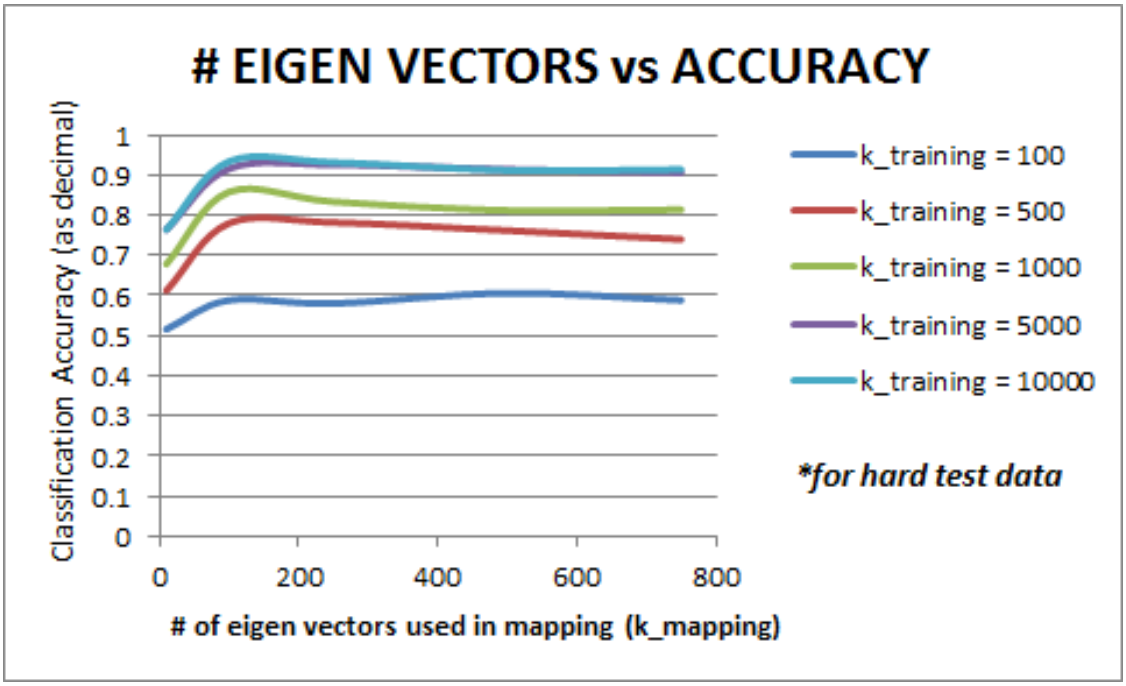
| | # top eigen vectors used in mapping | | | |
|---|---|---|---|---|
| k_training | top 100 | top 250 | top 500 | top 750 |
| **100** | 58.8% | 58.0% | 60.6% | 58.8% |
| **500** | 78.2% | 78.2% | 76.2% | 74.0% |
| **1000** | 85.8% | 83.4% | 81.2% | 81.4% |
| **5000** | 91.6% | 92.6% | 91.4% | 90.6% |
| **10000** | 93.4% | 93.2% | 91.2% | 91.4% |

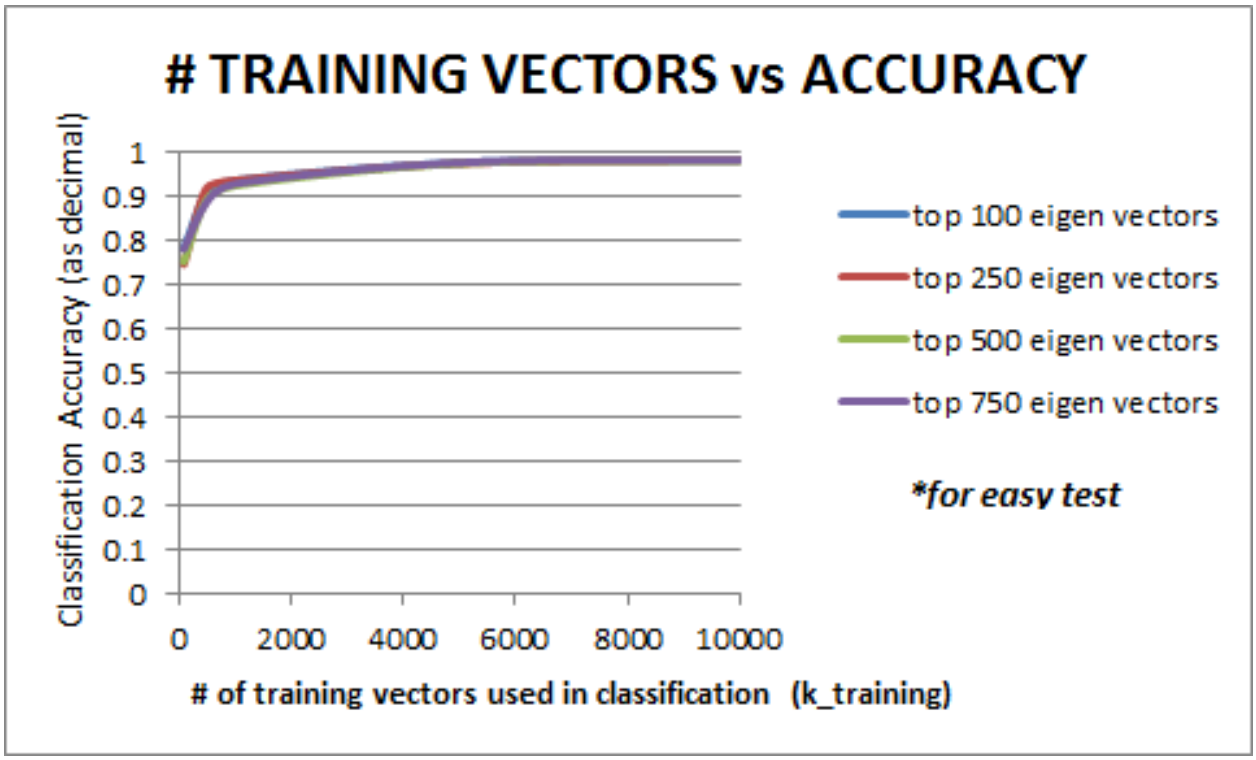
<< Hard Test Data

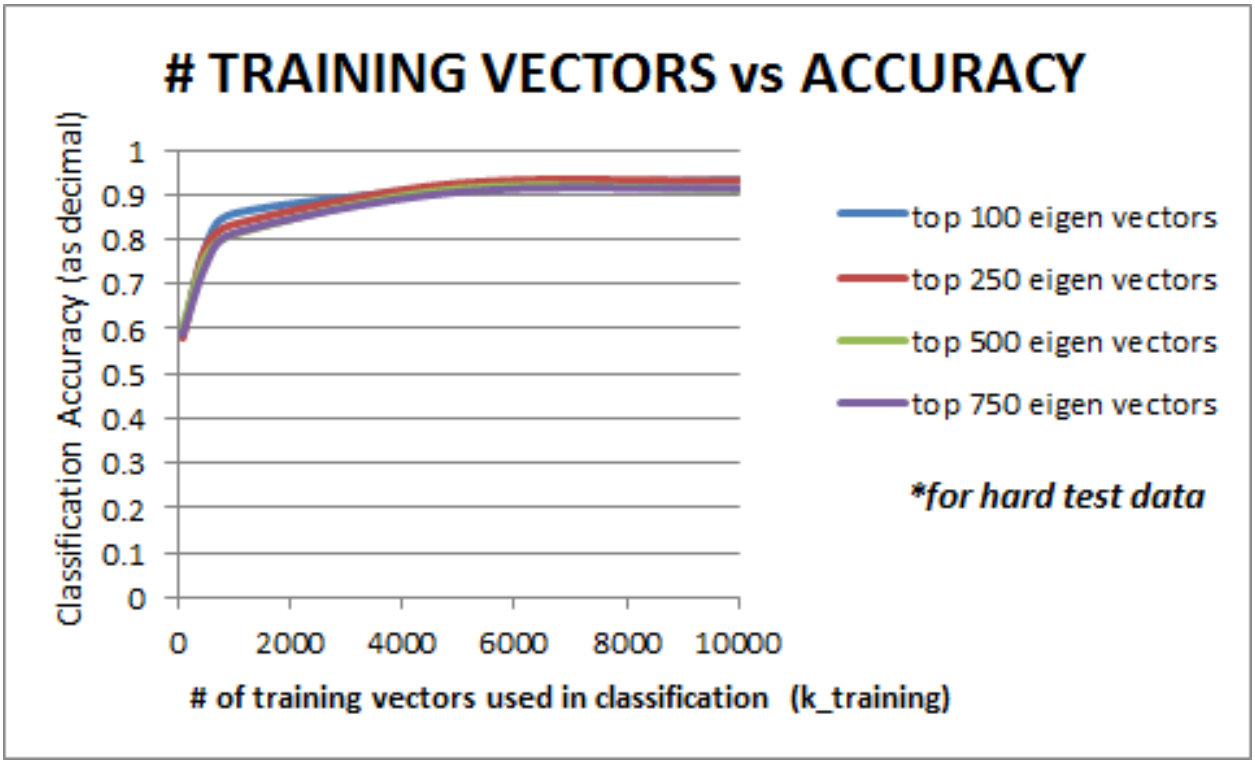
| Accuracy | # eigvecs (k_mapping) | # training vecs (k_training) | test vectors | easy/ hard |
|---|---|---|---|---|
| **95.0%** | 100 | 5000 | 100 | hard |
| **95.0%** | 100 | 5000 | 100 | hard |
| **93.8%** | 500 | 20000 | 1000 | hard |
| **93.8%** | 500 | 20000 | 1000 | hard |
| **93.4%** | 100 | 10000 | 500 | hard |
| **93.3%** | 500 | 20000 | 5000 | hard |
| **93.2%** | 250 | 10000 | 500 | hard |
| **93.2%** | 250 | 10000 | 500 | hard |
| **92.6%** | 250 | 5000 | 500 | hard |
| **91.6%** | 100 | 5000 | 500 | hard |

<< Hard Test Data




# TRAINING VECTORS vs ACCURACY
Classification Accuracy (as decimal)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2000
4000
6000
8000
10000
# of training vectors used in classification (k_training)
top 100 eigen vectors
top 250 eigen vectors
top 500 eigen vectors
top 750 eigen vectors
*for easy test


# TRAINING VECTORS vs ACCURACY
Classification Accuracy (as decimal)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2000
4000
6000
8000
10000
# of training vectors used in classification (k_training)
top 100 eigen vectors
top 250 eigen vectors
top 500 eigen vectors
top 750 eigen vectors
*for hard test data