AMATH 563 Homework 01
Regression, Model Selection, and Dynamic Mode Decomposition
Joseph J. Williams
2020 Apr 17

In this report, we present findings from a classification problem in which we train various algorithms to identify handwritten digits. We use the MNIST data set to train (N_{train} = 60,000) a variety of algorithms, including backslash, pseudoinverse, lasso, and elastic net, to identify a handwritten digit based only on its vectorized pixel representation. We use a variety of partitioning techniques, including partitioning randomly and partitioning based on digit; we also use a variety of training and testing techniques, including combinations of training and testing by separate digits. As initial cases, it is found that the backslash and pseudoinverse algorithms give fairly good results, and that partitioning the training or testing sets or training or testing digits separately only serves to decrease the success rates. Lasso and elastic net are found to be inferior to backslash and pseudoinverse algorithms.

Sec. I Introduction and Overview

Considering the origins of Kepler's laws of planetary motion, we see that data science has been a part of the scientific method almost from its inception. Carrying that tradition on, in this study, we will be working with the MNIST data set, which consists of tens of thousands of images of handwritten digits 0 through 9, as well as labels identifying what digit the image is supposed to represent. The data set is already divided into training and testing sets; we will explore different algorithms and strategies to develop a mapping from the image space to the label space for the training data and evaluate the quality of the mapping against the testing data.

From the MNIST data set, we have a training set of size N_{train} = 60,000 and a testing set of size N_{test} = 10,000. Every image also has a label associated with it, so we know what digit it is without having to manually look at the image. Our images are 28x28 pixels; thus, each image can be represented as a (28 x 28) matrix of values quantifying how dark that pixel is, from 1 (white) to 256 (black). We can then rearrange this matrix into a row vector of length 28^2 = 784 by lining up each row of the matrix. As long as we "unpack" the image vector back into the matrix in the same way that we packed it up, we can treat the vector as the pixel representation of the image.

Next, we assemble the row vectors of each of the N_{train} training images into a single matrix A_{train}, sized (60,000 x 784). Recalling the labels associated with each image, if we create for each image a unit row vector with a one in the entry corresponding to what digit it is and zero in all other entries, we can stack these row vectors and obtain the matrix B_{train}, sized (60,000 x 10). The (i,j) entry of this matrix corresponds to whether image *i* is digit *j* (where the tenth column represents 0): if (i,j) = 1, then image *i* represents digit *j*; otherwise, it doesn't. Thus we have built the system $A_{train} X_{train} = B_{train}$, with the unknown X_{train}, sized (784 x 10). Regarding the significance of X_{train}, each of the 10 columns are associated with a different digit, and the entries of the matrix represent the relative importance of each pixel (down the columns) for that digit (across the rows). Once we solve for X_{train}, we will use it to compute $B_{predict} = A_{test} * X_{train}$, which we will use to predict which digit a given image is and compare the results to B_{test}, the true labels of each of the images in A_{test}.

Note that, while all of our images are vectors in R^{784}, not every vector in R^{784} is one of our images. Specifically, imagine a 28x28 pixel image with black and white dots distributed randomly with random intensity. This would look nothing like a handwritten digit. So although the dimension of the space is large, we are interested in only a small, specific subspace. Further, this system is vastly overdetermined, with 60,000 equations (the training images) for just 784 unknowns (the intensities of the pixels). We are further abstracting the data by giving each image a label – its digit 0 through 9. The complexity the labels introduce is discussed later.

Returning to our overdetermined linear system $A_{train}\, X_{train} = B_{train}$, we will demonstrate several ways to solve this system, including different algorithms and different strategies to promote sparsity, and explore the nature of a successful solution by using X_{train} to interpret the testing data.

Sec. II Theoretical Background

Being a classification problem, its nature is subtle: we want not to minimize the error between the predicted and real pixel intensities (i.e. $B_{test} - B_{predict}$) but just to keep it below some value. Physically, we don't need the algorithm to be completely certain it's picking up a, say, 8, so long as it's more certain it's an 8 than it is any other digit. Thus, how well the algorithm does *absolutely* is less important than if it does well *enough*. This idea is discussed further later.

With regards to being overdetermined, we note that working with an underdetermined system would be nonsensical, as we'd be missing images of some digits and have no way to learn what pixels are important to that digit. Further, it would be unphysical for this system to have a unique solution. Having a unique solution would imply that there exist only a certain finite number of ways to write a certain digit, when in reality there are perhaps an infinitude of ways an individual can write any given symbol.

Mathematically, an exactly determined system would have a unique solution if the matrix A were square and full rank. Recall what the dimensions of A (60,000 x 784) signify for our case: for each of the 60,000 training images, there are certain intensities for each of the 784 pixels. A full rank system would have only as many images as pixels (and images identical to or a "multiple" of any other, i.e. a linearly independent system, as implied by A having full rank), and thus each weighting in X would have a unique value for a given B. Recalling the complexity introduced by the labels hinted at earlier, in our case, we are abstracting this data one level deeper by further associating with each image a certain label, namely, the ten digits 0 through 9. Now, there are only 10 unique labels for all 70,000 images. If we were identifying as many unique symbols as we had images, and further each image was unique and each symbol presented only once, then our system as a whole would be uniquely determined. However, this would have the limitation of being trained on only a single sample for each digit – any attempt at identifying a handwritten digit not exactly like the one it was trained on (or at least different enough that its finite pixel representation is not identical) would suffer some error. Thus we have reasoned ourselves into the natural conclusion (also suggested by Figure 4.1 of the textbook) that, when developing models for extrapolation, in order to minimize the error we must be over-fit rather than under-fit. Model complexity is discussed further later.

Sec. III Algorithm Implementation and Development

As discussed earlier, there are a variety of ways to solve AX = B. We will use two, backslash (bsl) and pseudoinverse (mps),[1] as introductory cases; as more advanced cases, we will use lasso, which offers a variety of parameters for us to tune. As considered earlier in this report, we are interested less in X picking up a digit *with certainty* and more in X picking up a digit with *enough* certainty. Thus, when considering what error metric to use and deciding on a heuristic by which to judge which scheme was the most successful, we convert the computed $A_{test} * X_{train} = B_{predict}$ into the "binary form" that B_{train} is in, with 0s everywhere except for 1s at the entries corresponding to a particular image representing a particular digit. This is done by replacing the maximum entry of each row with 1 and setting the rest to 0. However, there are other ways this might be done – for example, the entry with a value closest to 1 might be picked as representing the predicted digit. However, initial testing showed that this latter method and others almost never correctly predicted the digit; as such, they are not further discussed in this report.

Having obtained a binary $B_{predict}$, we may understand that the error ε is a vector whose entries are the error for each image in the test set; the values of the errors are either 0, if the prediction for that image was correct, or 1, if the prediction was incorrect. Said another way, we're interested in whether or not we got the digit right, not *how* right.

[1] **M**oore-**P**enrose p**S**eudoinverse... It seemed clever enough, at the time, but now, I wonder why I didn't just go with "inv." Alas.

Thus, the sum of the entries of ε is the total number of incorrectly identified images from the test set. We may now easily compute the success rate *SR* of the algorithm:

$$SR = 1 - sum(\varepsilon) / N_{test}.$$

Using backslash and pseudoinverse, we find that $SR_{bsl} = SR_{mps} = 0.8534$, meaning that the mappings X_{train} from the two algorithms correctly identify new handwritten digits with 85.34% accuracy. In order to improve this, we try k-folding. First, we partition the training dataset into k subsets, training k separate $X_{train,k}$ using both the backslash and pseudoinverse algorithms; ordinarily, one should randomize how the training data is partitioned, but here we assume that these images were randomized before being distributed. Next, we build the optimal X_{train} from the k-different $X_{train,k}$. In this, there is significant user choice. One obvious way would be to average the k different $X_{train,k}$; another reasonable, though more laborious, idea would be to pick the values from the different $X_{train,k}$ that lead to the lowest error w.r.t. some norm. We opt for the former method for simplicity; proceeding with the analysis, reference Fig. 3.1 below, which shows SR as a function of k for the two algorithms used so far.

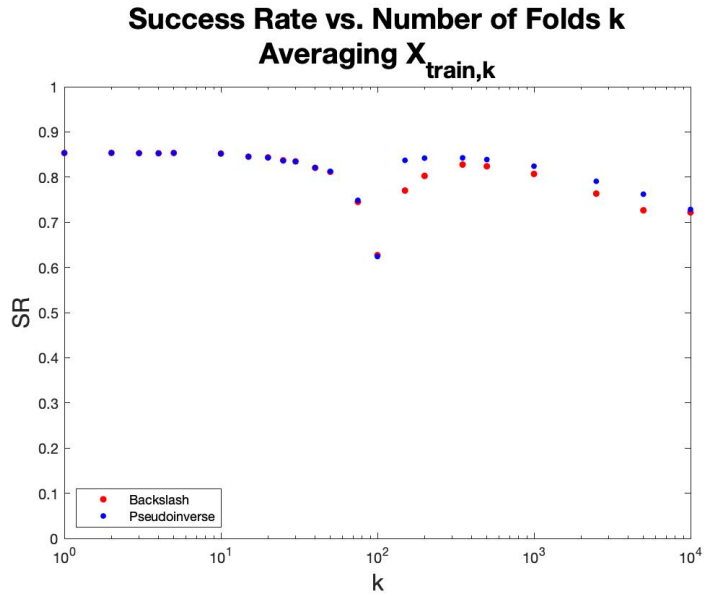

Fig. 3.1: Success Rate vs. Number of Partitions, Two Algorithms

We see very similar behavior in the backslash and the pseudoinverse solutions, especially for $k < 10^2$. For $k > 10^2$, backslash appears to have a slightly lower success rate than pseudoinverse. They both also share the same curious behavior: the success rate plummets as $k \rightarrow 100^-$ but rises again almost to its maximum value for $k = 500$ before slowly decreasing as $k \rightarrow N_{train}$. A priori, we might expect that partitioning and combining the results in some way would lead to better results, perhaps through arguments about averaging out the noise. In practice, this is not what we find, suggesting that using a single subset is the optimal strategy for this data i.e. that the backslash and pseudoinverse algorithms are already fairly robust without any partitioning. With ~6000 examples of each digit, and only 784 pixels per image, we've seen almost every way there is to write a given digit at this resolution. With this, we hope to avoid overfitting our model.

Note that, as we change the degree and type (discussed in § IV, shortly) of folding, the entries of X_{train} and $X_{train,k}$ change, reflecting different weightings as a result of trainings on different subsets of data; however, the sizes of X_{train} and $X_{train,k}$ (784 x 10) are unchanged, reflecting that different folding schemes don't change the type of information encoded. The folding essentially takes a vastly overdetermined system and breaks it up into many smaller and more moderately overdetermined systems, each with its own optimal solution; we then combine these solutions.

Sec. IV Computational Results

Given the context of our problem, a more clever way of partitioning the data would be to divide the training data by label; that is, we will generate $x_{train,d}$, a vector sized (784 x 1) for each digit *d* from 0 through 9. We can train each $x_{train,d}$ with the entire training image space or just the image space with the relevant label. Then we can either:

1. create an X_{train} matrix with the column vectors $x_{train,d}$, and use that to compute $A_{test} * X_{train} = B_{pred}$; or
2. evaluate $A_{test,d} * x_{test,d} = b_{pred,d}$ for each digit d.

Regarding option 1.), this analysis is similar to what we did previously. We can further subdivide each digit's partition into d_k subpartitions and reproduce Fig. 3.1 above but with the additional initial partitioning by digit. See Fig. 4.1 below, which shows SR vs. k for this method, with $x_{train,d}$ derived the two ways suggested previously; note that here, k is the number of partitions after the initial partition by digit.

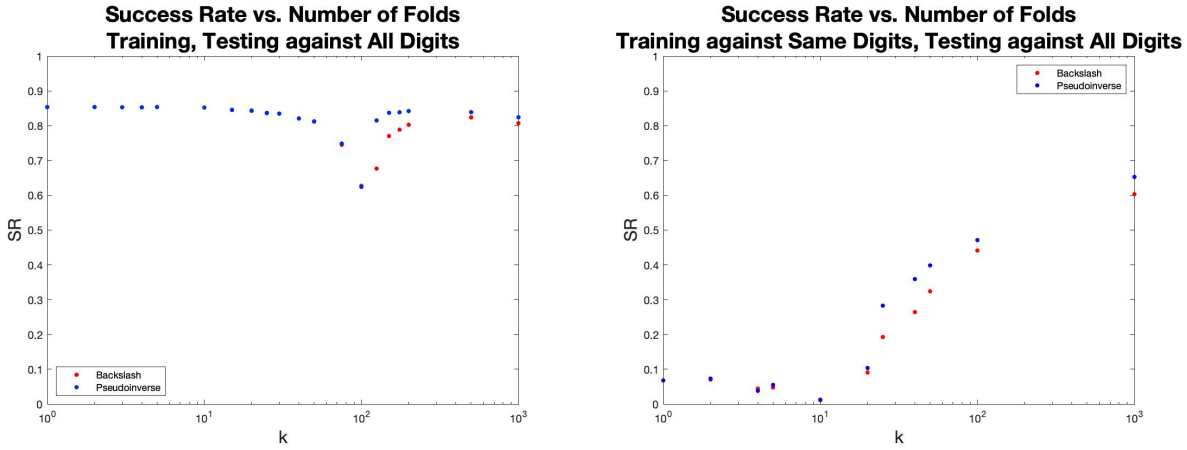

Fig. 4.1: Success Rate vs. Number of Partition; two algorithms, testing against all digits
Left: Training against all digits; *Right*: Training against only the same digit

When initially partitioning by digit, we see that, when we both train for all digits and test the algorithm against all digits (left), the behavior is almost exactly the same as that displayed in Fig. 3.1: the success rate is greatest when k = 1, and as $k \to 100^-$, the success rate for both algorithms plummets, but rises again almost to its peak value as k increases past 100, and then slowly decreases for large k, suggesting again that backslash and pseudoinverse are already fairly robust without any partitioning. The difference between the backslash and pseudoinverse algorithms seems also to be very similar. Note, however, that the decrease in the success rate as k becomes large is much slower here than in Fig. 3.1; when training for all digits and testing against all digits, this seems to be the only benefit of initially partitioning by digit. Note also that the maximum success rate here is 85.34% – the same as in Fig. 3.1

However, when we train only one digit at a time, and then test against all digits, we see something very different, namely a much lower success rate for all values of k. The dependence of the success rate on k is also different: it appears that, for both backslash and pseudoinverse, once $k \sim 10$, the success rate almost increases monotonically with k – something not previously observed. For $k < 10$, there doesn't seem to be a clear relationship between success rate and k. This suggests that testing the digits individually is a much weaker strategy than testing them altogether. One reason behind why this might be is because of the similarity between many digits: an 8, if drawn poorly or faintly enough, may easily appear to be a 2, 3, 5, 6, 9, or 0.[2] Thus, when we train the digits altogether, not only does the algorithm learn what an 8 looks like, it also learns what an 8 *doesn't* look like. Training the digits separately robs the algorithm of this feature, leading to a much weaker identification algorithm overall and thus much lower success rates. Consequently, since the algorithm was much weaker to begin, we observe that partitioning is actually effective (as k increases past $k \sim 10$, so does the success rate).

[2] At least, that's as far as my imagination (and squinting) can take me.

Returning to the two options laid out previously, option 2.) may seem trivial: We just trained $x_{train,d}$ on a few thousand samples for each digit – won't identifying new handwritten digits be a slam dunk? Alas, there is hope for many hopeless things in this forlorn world. Unfortunately, while training on a vast number of samples may increase our confidence in our ability to correctly identify a new image, unless we test as many samples as we have pixels, we won't be able to identify a new image with perfect success. See Fig. 4.2 below, which shows SR vs. k for this method, with $x_{train,d}$ again derived two ways.

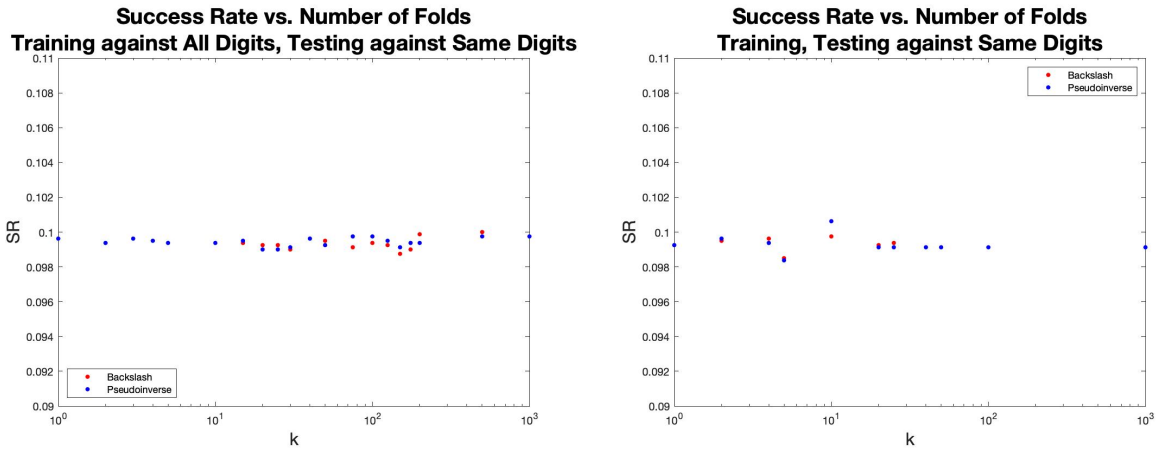

Fig. 4.2: Success Rate vs. Number of Partitions; two algorithms, testing against only same digit
Left: Training against all digits; *Right*: Training against only the same digit

What we see above is much different from what we have seen previously; note the different axes on both figures, which show that the success rates are much lower for all values of k. While the success rates as a function of k begin similarly to how Fig. 4.1 (right) did, with only a haphazard relationship between success rate and k, we see that, as k increases, the success rate changes little. This suggests that testing the digits individually, irrespective of how the algorithm was trained or the training and testing data partitioned, is a catastrophically poor choice.

We offer some physical intuition: When writing a particular digit by hand, there are many small (and large) differences that could appear from one instance to the next. Given that we obtained such a low success rate when testing the digits individually, what this suggests is clear: When testing the digits altogether, these small differences essentially become noise that the algorithm is able to ignore, but when testing digits individually, the algorithm is essentially holding each digit to a much higher standard. If the testing digit doesn't look exactly or almost exactly like what the algorithm was trained on, then it won't correctly identify it. It could also just be a bug.

This concludes our introductory explorations with backslash and pseudoinverse; we now move to more advanced methods, namely lasso, which has a variety of parameters available for tuning. The two main relevant parameters are λ, the value of the regularization coefficients, and α, a parameter unique to MATLAB's lasso(X,y) command that controls the weighting of lasso vs. ridge optimizations; reference the table below. We will be testing a variety of values of α and λ together. Fig 4.3 on the following page shows the success rate vs. λ for various values of α.

$\alpha = 1$	Lasso
$0 < \alpha < 1$	Elastic Net (Weighted between lasso and ridge)
$\alpha = 0$	Ridge

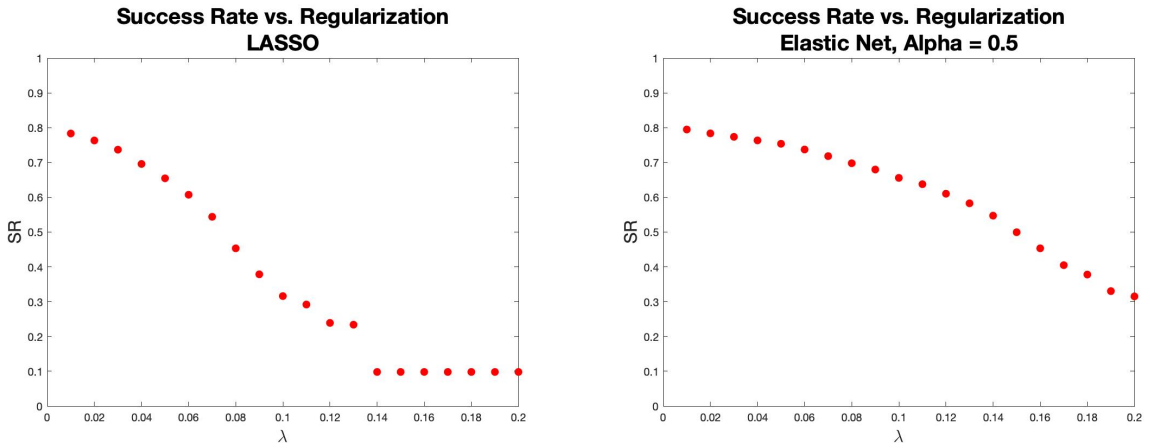

Fig 4.3: Success Rate vs. λ
Left: α = 1; *Right*: α = 0.5

Note that all plots above were created without any partitioning or special training or testing schemes. We see that, in general, as λ increases, the success rate decreases; however, this effect is slower for $\alpha < 1$, and it is hypothesized that as $\alpha \to 0$ (i.e. as the optimization becomes less like lasso and more like ridge), the success rate will decrease ever more slowly. Curiously, when $\alpha = 1$, the success rate bottoms out at $\lambda = 0.13$ and remains at this minimum value of success rate for all values of λ up to at least $\lambda = 1$. However, no such behavior is observed when $\alpha = 0.5$. Finally, note that the success rates are universally lower than the success rates for backslash and pseudoinverse obtained early in this study, suggesting that, at least for this data set, lasso, elastic net, and ridge are inferior algorithms.

Sec. V Summary and Conclusions

In this report, we have attempted to train various algorithms to identify handwritten images. We used the MNIST data set, which consists of tens of thousands of handwritten digits and their associated labels (serving as identifiers). Various algorithms, including backslash, pseudoinverse, lasso, and elastic net, were used alongside various partitioning, training, and testing strategies.

In general, it was found that the backslash and pseudoinverse algorithms, in the absence of any partitioning and without any special training or testing schemes, have superior performance. As we introduce various partitioning schemes, including partitioning randomly and partitioning by digit, the success rate is either not materially affected or it is decreased. Further, as we introduce various training and testing schemes, including combinations of training and testing only against certain digits, the success rate drops, sometimes substantially and apparently without any hope of it recovering. Regarding the more advanced algorithms, including lasso, and elastic net, although they are generally more complex algorithms with more parameters to tune, with the MNIST data set, their performances are inferior to those of backslash and pseudoinverse.

Topics for future work include:

- A more exhaustive study of success rate vs. k when using backslash and pseudoinverse algorithms, especially for $k \sim 10^2$. Since this appears to be a critical value of sorts, a better understanding of the success rate's behavior for $k \sim 10^2$ would better our understanding of these training algorithms as a whole.
- A more exhaustive study of success rate vs. λ and of success rate vs α. In particular, a plot of success rate vs. α would be of great interest, as would more plots of success rate vs. λ for a variety of values of α.
- Partitioning, training, and testing schemes in conjunction with lasso, elastic net, and ridge. Although such schemes were found to be detrimental for backslash and pseudoinverse, their effects on lasso, elastic net, and ridge were not explored in this study.

Appendix A

This appendix provides a brief explanation of how the code runs.

First, run the first two sections of AMATH563_HW01_Main.m. This loads the data and creates the matrices that will be used to train and test the algorithms.

Next, run one of the following three sections of the main code:

- %% Section III – Initial Solve and Initial Partitioning
- %% Section IV - Partitioning by Digit
- %% Section V - Advanced Solves

This will run the .m files listed in those sections and execute the various algorithms with their various partitioning, training, and testing schemes.

Appendix B

See GitHub repository.