HW1: MNIST Exploration

Alexander Van Roijen

May 3, 2019

I Introduction and Overview

The MNIST[1] data set has been a subject of many great publications and even more classroom assignments. It is the basis for many introductory machine learning courses and thus offers many great resources for education. I explored how two particular python library packages can handle this image classification task[6],[7]. We will see that simple binary classification models out preform the batch trained linear models in this task, and that Regularized Least Squares(RLS) tends to preform better than lasso models. The caveat being that the penalty term within RLS must be present to avoid sparsity induced errors.

II Theoretical Background

In general, we can phrase many different quantitative and qualitative problems into an $Ax = b$ form. There are three scenarios here.

1. **Consistent System** This occurs when we have a square matrix, or exactly as many equations as unknowns, and thus have one unique solution.
2. **Underdetermined** This occurs when we have more unknowns than equations,i.e. more columns than rows. This means we have infinite solutions.
3. **Overdetermined** This occurs when we have more equations than unknowns, i.e. more rows than columns. Now we have no exact solutions

In general, most problems we tackle in reality are of the overdetermined form. However, you still see problems of the underdetermined form in many biological problems where we have small sample sizes.

Since we are looking at handling the MNIST problem, we are dealing with an overdetermined system. However, we said there were no exact solutions. The keyword **exact** meaning we are going to need to find an x that satisfies some alternative condition. In particular I look at minimizing the following three models.

- Regularized Least Squares
  $O(x, \alpha) = \|b - Ax\|_2^2 + \alpha \|x\|_2^2$ (1)
- Least Squares
  $O(x) = \|b - Ax\|_2^2$ (2)
- Lasso
  $O(x, \alpha, n) = \frac{1}{2n} \|b - Ax\|_2^2 + \alpha \|x\|_1$ (3)

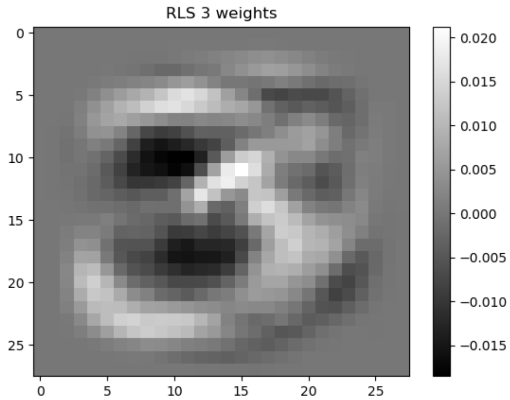
The primary difference can be seen in their penalty term. Regularized Least Squares squares the weights, while Lasso simply adds up their absolute value. Meanwhile, standard least squares makes no constraint on the size of our weights. We will see later that Lasso promotes more zero values, or **sparsity**, meanwhile regularized least squares allows for a lot of non zero values.

III Algorithm Implementation and Development

More formally, we will be solving $Ax = b$ with the following parameters. $\boldsymbol{A = n \times p}$, $\boldsymbol{b = n \times d}$ where $\boldsymbol{n = 60'000}$ for the training data, $\boldsymbol{p = 28 * 28 = 784}$, and $\boldsymbol{d = 10}$ where d is the vector notation representing our label. This means $\boldsymbol{x = p \times d}$. Essentially, each column will represent a weight vector w that will convert an image into a "probability" of it being that number.Meanwhile when we explore what happens when we solve $Ax = b$ with binary output our d will change to 1. This new 1-d column will represent the presence of the numbers label or lack thereof. The resulting x still represents the same weight vector, but now only has to solve a simpler case.

To solve these problems, I utilize pythons sklearn libraries of lasso and ridge regression[(6],[7]. ***ridge*** and ***least squares*** uses **Singular Value Decomposition (SVD)** to find its solution, meanwhile ***lasso*** uses **coordinate descent** to find its solution. The penalties for the algorithms and corresponding results were determined by leveraging the structure of the equations in conjunction with a trial and error process. For (3), the residuals are normalized by a factor of the sample size, thus the orders of magnitude would vary in the order of 10^4. The trial and error came in the form of examining Mean Squared Error(MSE)and graphical results of our weights (figure 1).

Figure 1: RLS penalty 10000 batch mode weight vector for 3. Its encouraging to see weights that closely resemble a 3



In the binary output classification problem, the dimension of our x is altered by a magnitude of 10^1 as we have reduced d to 1. Consequently, when handling the binary classification task, I reduced the penalty by a similar order of magnitude. There can be more thorough work done to better choose a penalty (in fact, cross validation would be one), but I did not focus too much on this aspect. Find the penalties in the table below.

batch	model	penalty
False	LST_SQR	0.000
False	RLS	1000.000
False	lasso	0.010
True	LST_SQR	0.000
True	RLS	10000.000
True	lasso	0.001

Table 1: Penalties for corresponding models and objectives. *batch* represents if we are solving the d=10 problem (true) or not.

To select the most important pixels and encourage sparsity, I used the following algorithm.

Algorithm 1:
Sort a (weight,location) pair vector $\langle x, y\rangle$ in descending order of absolute weight value $|x_i|$. Pop the top value of the list x_1, y_1 and record its location. Accumulate the percentage of coverage as follows : $percentCover \mathrel{+}= \frac{|x_1|}{sum(x)}$. Repeat this process until either $percentCover \geq 0.90$ or we have exhausted $\langle x, y\rangle$. Then the locations recorded are used to filter the original vector x to its sparse form x'.

The reason for using 90 percent is too better allow sparsity in both lasso and RLS as a simple 'top x' rule would favor lasso over RLS and create biased results. Trial and error was used to determine a reasonable percentage that introduced a fair amount of sparsity. It is important to note that accuracy was not looked at to determine this metric, unlike the penalty.

Lastly, to measure accuracy, **Classification Error (CE)** was used. Here **CE**$= 1 - \frac{\#\text{correctly identified}}{\text{total possible}}$. In more detail, given a weight vector x, input A, and output b, we do the following.
Algorithm 2:
For each row b_i of output b, we check the corresponding row of $A \cdot x$, call it c_i. If the index of the maximum value of c_i is the same as the index of 1 in b_i, we increment the correctly identified counter for that particular number. This gives us a seperate CE for each individual number and lets us make easier comparisons between all models and cases when $d = 10$ versus $d = 1$

IV Computational Results

Batch	model	penalty	Avg_Train_CE	Avg_ Test_CE
False	RLS	0.000	0.054267	0.055160
False	RLS	1000.000	0.055473	0.054840
False	lasso	0.010	0.093370	0.092900
True	RLS	0.000	0.197857	0.200308
True	RLS	10000.000	0.218289	0.213300
True	lasso	0.001	0.207368	0.202691

Table 2: Showcases performance of **full** models with average Classification Errors(CE) on training and test data with various penalties, and whether or not the training was done on individual or batch outputs

Batch	model	penalty	Avg_Train_CE	Avg_Test_CE
False	RLS	0.000	0.100028	0.100090
False	RLS	1000.000	0.056463	0.055680
False	lasso	0.010	0.096792	0.096910
True	RLS	0.000	0.900017	0.900000
True	RLS	10000.000	0.219919	0.215081
True	lasso	0.001	0.222075	0.217960

Table 3: Showcases performance of **sparsity** induced models with average Classification Errors(CE) on training and test data with various penalties, and whether or not the training was done on individual or batch outputs

All results can be found succinctly represented in tables 2 and 3, but lets tackle this in order. First, we can find in table 2 the results of our experiments for the full models without filtering out the most important pixels according to algorithm 1 above. The average classification errors listed are averaged over all numbers under the penalty and model listed. The *Batch* boolean states if we trained our model with $d = 1$ ($Batch =$ False) or not. We can see that our error is lower when we simplify the classification task and train separate models on the entire dataset. This makes sense as any weight vector x_i for some number i would not need to fight for the importance of various pixels with the other weight vectors in the minimization problem.

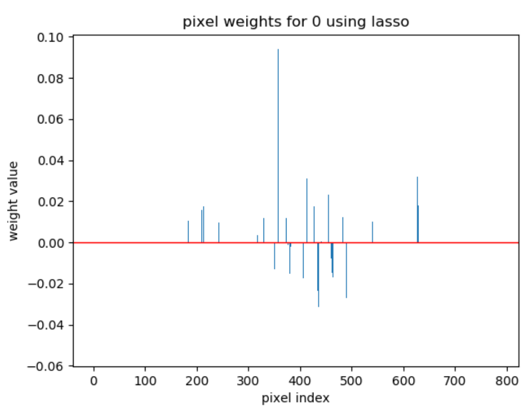


Figure 2: bar graph representing weights of pixel values on indepdently trained x for 0

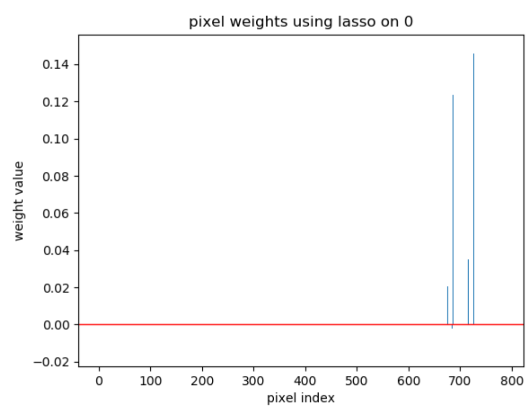


Figure 3: bar graph representing weights of pixel values on batch trained x for 0

Figure 2 showcases this situation nicely. As you can see, there is more variety and less extremes in the binary classification weights. This is due to that lack of 'fighting' for unique pixel values that differentiate themselves from other numbers. We could imagine that pixels in the number 6 may compete with the same pixels in the weight vector for 0.

Now when we compare tables 2 and 3,representing full and sparse models respectively, we can see some interesting differences. First, we can see that all our CE are higher in the sparsity induced model for both training and testing. However, this is to be somewhat expected, particularly on the training set as we have taken away weights that the model deemed were significant. The most notable of all however is the large change in the accuracy of the ordinary least squares fit (RLS with penalty 0). We can see a loss of 5% accuracy on the independently trained data and a huge loss of about 70% on the batch trained data! Luckily, we can explain this. Due to $\alpha = 0$, RLS will fit a model with no regard to the magnitude of our x. We can see in figure 5 that the scale of the weight vector for 5 can go orders of magnitude higher than we saw in other vectors. Now when we consider what happens when we use **(1)** to select the most important pixels, we can understand how the classifications become so extreme as we now have limited ourselves to only a few high value pixels to identify the correct number.

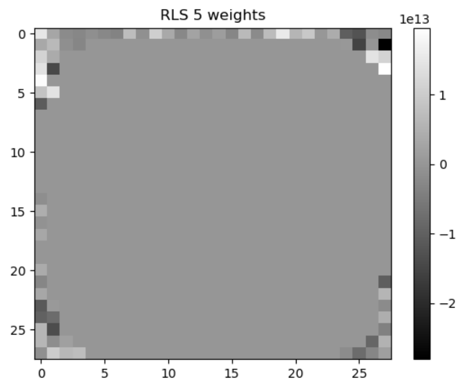


Figure 4: bar graph representing weights of pixel values on indepdently trained x

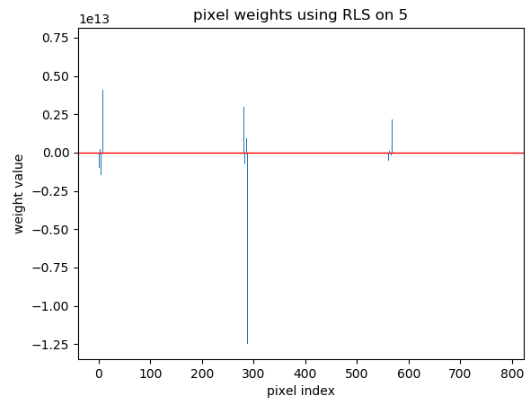


Figure 5: bar graph representing weights of pixel values on batch trained x

We can clearly tell that the weights in figure 4 arent picking up on the discernible features of the model like we saw in figure 1. Instead it is picking up on the noise and maximizing the values there in order to properly classify the data.

V Summary and Conclusions

Overall, we have seen how well multiple solvers find weights for the hand written character classification task. We used purely linear solvers that were trying to determine weight vectors that would convert the pixels from a 28×28 image into a vector of probabilities. We conclude that the solvers for simple binary classification outperformed batch trained models, but obviously address a different question. Further, we see that with the given penalties, RLS appears to outperform our lasso models but are more sensitive to the penalty term.

VI Future Work

I think the further benefit to this exploration is the ability to quantify our intuition in our results and begin to establish a "gut feeling" for how other problem sets will be solved by our linear models. We can see many of the issues with unpenalized RLS and sparsity make our models less generalizable. I aim to apply similar thoughts when addressing other problems within my field of work.

VII References

- 1: Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.
- 2: Scikit-learn. Lasso. April 15, 2019. Electronic. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html
- 3: Numpy. P-Inv. April 15, 2019. Electronic. https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.pinv.html
- 4: Numpy. Least Squares. April 15, 2019. Electronic. https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.lstsq.html
- 5: Numpy. P-Norm. April 15, 2019. Electronic. https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.norm.html
- 6: Scikit-learn. Ridge. April 15, 2019. Electronic. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html
- 7: Scikit-learn. Lasso. April 15, 2019. Electronic. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

VIII Results Appendix

More code,images, and results can be found on my github.

	Number	Train_MSE	Train_CE	Test_MSE	Test_CE	model	penalty	indiv
0	0	0.000984	0.071800	2.418665e-03	0.070200	lasso	0.010	False
1	0	0.000876	0.033700	2.148676e-03	0.036200	RLS	1000.000	False
2	1	0.001035	0.112433	2.535835e-03	0.113600	lasso	0.010	False
3	1	0.001219	0.063017	2.973825e-03	0.061200	RLS	1000.000	False
4	2	0.000976	0.092333	2.419239e-03	0.095100	lasso	0.010	False

5	2	0.000824	0.045267	2.038653e-03	0.047000	RLS	1000.000	False
6	3	0.001002	0.100383	2.409180e-03	0.098300	lasso	0.010	False
7	3	0.000865	0.046917	2.086586e-03	0.045200	RLS	1000.000	False
8	4	0.001089	0.097367	2.679112e-03	0.098200	lasso	0.010	False
9	4	0.001120	0.080533	2.778384e-03	0.081700	RLS	1000.000	False
10	5	0.001147	0.090350	2.787923e-03	0.089200	lasso	0.010	False
11	5	0.001190	0.084317	2.888626e-03	0.080500	RLS	1000.000	False
12	6	0.000895	0.080167	2.169961e-03	0.075900	lasso	0.010	False
13	6	0.000762	0.029450	1.855866e-03	0.030100	RLS	1000.000	False
14	7	0.001107	0.102650	2.697607e-03	0.101000	lasso	0.010	False
15	7	0.001003	0.054733	2.463914e-03	0.054600	RLS	1000.000	False
16	8	0.001074	0.087067	2.625834e-03	0.086600	lasso	0.010	False
17	8	0.001034	0.045600	2.535932e-03	0.043600	RLS	1000.000	False
18	9	0.001084	0.099150	2.659122e-03	0.100900	lasso	0.010	False
19	9	0.000945	0.071200	2.301098e-03	0.068300	RLS	1000.000	False
20	0	0.000855	0.094716	2.098120e-03	0.081633	lasso	0.001	True
21	1	0.001164	0.144616	2.841497e-03	0.148018	lasso	0.001	True
22	2	0.000835	0.214837	2.061320e-03	0.228682	lasso	0.001	True
23	3	0.000878	0.173870	2.116556e-03	0.147525	lasso	0.001	True
24	4	0.001093	0.283978	2.707229e-03	0.272912	lasso	0.001	True
25	5	0.001143	0.606346	2.769706e-03	0.602018	lasso	0.001	True
26	6	0.000759	0.089219	1.854691e-03	0.099165	lasso	0.001	True
27	7	0.001019	0.217239	2.493921e-03	0.215953	lasso	0.001	True
28	8	0.001059	0.076055	2.592882e-03	0.059548	lasso	0.001	True
29	9	0.000945	0.172802	2.296413e-03	0.171457	lasso	0.001	True
30	0	0.000831	0.082728	2.039014e-03	0.062245	RLS	10000.000	True
31	1	0.001329	0.179472	3.266554e-03	0.181498	RLS	10000.000	True
32	2	0.000848	0.201074	2.090995e-03	0.207364	RLS	10000.000	True
33	3	0.000890	0.168488	2.142691e-03	0.147525	RLS	10000.000	True
34	4	0.001076	0.283978	2.660063e-03	0.267821	RLS	10000.000	True
35	5	0.001201	0.710754	2.915327e-03	0.719731	RLS	10000.000	True
36	6	0.000832	0.088712	1.998003e-03	0.086639	RLS	10000.000	True
37	7	0.001073	0.200319	2.621824e-03	0.214981	RLS	10000.000	True
38	8	0.001045	0.056742	2.554888e-03	0.040041	RLS	10000.000	True
39	9	0.000991	0.210624	2.413468e-03	0.205154	RLS	10000.000	True
40	0	0.000899	0.033267	1.375928e+08	0.036700	RLS	0.000	False
41	1	0.001282	0.068233	4.623389e+07	0.065900	RLS	0.000	False
42	2	0.000803	0.041217	2.022587e+08	0.044900	RLS	0.000	False
43	3	0.000872	0.043133	2.029217e+08	0.041900	RLS	0.000	False
44	4	0.001200	0.083117	4.152722e+08	0.088400	RLS	0.000	False
45	5	0.001213	0.082833	1.400219e+08	0.081300	RLS	0.000	False
46	6	0.000733	0.028650	9.876888e+07	0.030900	RLS	0.000	False
47	7	0.001008	0.049550	8.646927e+07	0.052400	RLS	0.000	False
48	8	0.001053	0.047917	2.289615e+08	0.047200	RLS	0.000	False

49	9	0.000948	0.064750	2.503748e+08	0.062000	RLS	0.000	False
50	0	0.000899	0.098768	1.375928e+08	0.087755	RLS	0.000	True
51	1	0.001282	0.133492	4.623389e+07	0.151542	RLS	0.000	True
52	2	0.000803	0.193353	2.022587e+08	0.231589	RLS	0.000	True
53	3	0.000872	0.139618	2.029217e+08	0.124752	RLS	0.000	True
54	4	0.001200	0.302123	4.152722e+08	0.283096	RLS	0.000	True
55	5	0.001213	0.574802	1.400219e+08	0.581839	RLS	0.000	True
56	6	0.000733	0.094289	9.876888e+07	0.107516	RLS	0.000	True
57	7	0.001008	0.195052	8.646927e+07	0.193580	RLS	0.000	True
58	8	0.001053	0.080328	2.289615e+08	0.073922	RLS	0.000	True
59	9	0.000948	0.166751	2.503748e+08	0.167493	RLS	0.000	True

Table 4: Full Results of all numbers, models, penaltys, and batch versus binary classification models. Table generated using pandas.to_latex() along with the long table package

	Num	Train_MSE	Train_CE	Test_MSE	Test_CE	model	penalty	indiv
0	0	1.050806e-03	0.086267	2.581412e-03	8.650000e-02	lasso	0.010	False
1	0	8.577602e-04	0.032350	2.106849e-03	3.370000e-02	RLS	1000.000	False
2	1	9.729781e-04	0.112367	2.373990e-03	1.135000e-01	lasso	0.010	False
3	1	1.168254e-03	0.060267	2.847191e-03	5.750000e-02	RLS	1000.000	False
4	2	9.905606e-04	0.095967	2.456941e-03	9.920000e-02	lasso	0.010	False
5	2	8.206486e-04	0.045783	2.031231e-03	4.760000e-02	RLS	1000.000	False
6	3	1.023162e-03	0.102183	2.461764e-03	1.010000e-01	lasso	0.010	False
7	3	8.621103e-04	0.045417	2.079478e-03	4.400000e-02	RLS	1000.000	False
8	4	1.111922e-03	0.097367	2.734018e-03	9.820000e-02	lasso	0.010	False
9	4	1.172317e-03	0.087633	2.909531e-03	8.950000e-02	RLS	1000.000	False
10	5	1.138581e-03	0.090350	2.765175e-03	8.920000e-02	lasso	0.010	False
11	5	1.214522e-03	0.086033	2.951396e-03	8.270000e-02	RLS	1000.000	False
12	6	9.036041e-04	0.084967	2.196825e-03	8.240000e-02	lasso	0.010	False
13	6	7.894687e-04	0.032483	1.918578e-03	3.230000e-02	RLS	1000.000	False
14	7	1.136033e-03	0.104417	2.774172e-03	1.028000e-01	lasso	0.010	False
15	7	9.860542e-04	0.055900	2.418361e-03	5.550000e-02	RLS	1000.000	False
16	8	1.067965e-03	0.094883	2.610071e-03	9.540000e-02	lasso	0.010	False
17	8	1.025039e-03	0.045400	2.511167e-03	4.310000e-02	RLS	1000.000	False
18	9	1.087050e-03	0.099150	2.667606e-03	1.009000e-01	lasso	0.010	False
19	9	9.411909e-04	0.073367	2.288812e-03	7.090000e-02	RLS	1000.000	False
20	0	9.524048e-04	0.135911	2.330733e-03	1.244898e-01	lasso	0.001	True
21	1	1.051591e-03	0.134085	2.555184e-03	1.400881e-01	lasso	0.001	True
22	2	8.609481e-04	0.252769	2.135250e-03	2.606589e-01	lasso	0.001	True
23	3	8.874585e-04	0.189366	2.139349e-03	1.603960e-01	lasso	0.001	True
24	4	1.090839e-03	0.328483	2.702033e-03	3.126273e-01	lasso	0.001	True
25	5	1.118402e-03	0.617045	2.706667e-03	6.221973e-01	lasso	0.001	True
26	6	7.258331e-04	0.080940	1.793855e-03	9.185804e-02	lasso	0.001	True

27	7	9.953076e-04	0.262729	2.432412e-03	2.645914e-01	lasso	0.001	True
28	8	1.064971e-03	0.084943	2.608140e-03	7.186858e-02	lasso	0.001	True
29	9	9.483372e-04	0.134476	2.297788e-03	1.308226e-01	lasso	0.001	True
30	0	8.298152e-04	0.084754	2.035461e-03	6.734694e-02	RLS	10000.000	True
31	1	1.314861e-03	0.181104	3.231631e-03	1.823789e-01	RLS	10000.000	True
32	2	8.609420e-04	0.214669	2.124514e-03	2.257752e-01	RLS	10000.000	True
33	3	8.916444e-04	0.169793	2.146926e-03	1.524752e-01	RLS	10000.000	True
34	4	1.067933e-03	0.284320	2.642114e-03	2.708758e-01	RLS	10000.000	True
35	5	1.200657e-03	0.717211	2.915730e-03	7.219731e-01	RLS	10000.000	True
36	6	8.139944e-04	0.082291	1.957270e-03	8.350731e-02	RLS	10000.000	True
37	7	1.072352e-03	0.199521	2.617310e-03	2.071984e-01	RLS	10000.000	True
38	8	1.050354e-03	0.057084	2.566070e-03	4.106776e-02	RLS	10000.000	True
39	9	9.864431e-04	0.208438	2.399568e-03	1.982161e-01	RLS	10000.000	True
40	0	5.980658e+06	0.098750	1.375928e+08	9.810000e-02	RLS	0.000	False
41	1	1.183473e+07	0.112417	3.740208e+07	1.136000e-01	RLS	0.000	False
42	2	1.204904e+07	0.099333	2.140463e+08	1.033000e-01	RLS	0.000	False
43	3	3.010286e+06	0.102200	2.159510e+08	1.010000e-01	RLS	0.000	False
44	4	1.767464e+07	0.097400	4.152722e+08	9.830000e-02	RLS	0.000	False
45	5	1.268916e+07	0.090383	1.346115e+08	8.930000e-02	RLS	0.000	False
46	6	1.277374e+07	0.098650	8.967438e+07	9.590000e-02	RLS	0.000	False
47	7	1.156823e+07	0.104433	7.870132e+07	1.029000e-01	RLS	0.000	False
48	8	7.559770e+06	0.097550	2.289615e+08	9.750000e-02	RLS	0.000	False
49	9	5.082330e+06	0.099167	2.503748e+08	1.010000e-01	RLS	0.000	False
50	0	5.980658e+06	0.000169	1.375928e+08	-1.021405e-14	RLS	0.000	True
51	1	1.183473e+07	1.000000	3.740208e+07	1.000000e+00	RLS	0.000	True
52	2	1.204904e+07	1.000000	2.140463e+08	1.000000e+00	RLS	0.000	True
53	3	3.010286e+06	1.000000	2.159510e+08	1.000000e+00	RLS	0.000	True
54	4	1.767464e+07	1.000000	4.152722e+08	1.000000e+00	RLS	0.000	True
55	5	1.268916e+07	1.000000	1.346115e+08	1.000000e+00	RLS	0.000	True
56	6	1.277374e+07	1.000000	8.967438e+07	1.000000e+00	RLS	0.000	True
57	7	1.156823e+07	1.000000	7.870132e+07	1.000000e+00	RLS	0.000	True
58	8	7.559770e+06	1.000000	2.289615e+08	1.000000e+00	RLS	0.000	True
59	9	5.082330e+06	1.000000	2.503748e+08	1.000000e+00	RLS	0.000	True

Table 5: Sparse Results(implemented algorithm 1) of all numbers, models, penaltys, and batch versus binary classification models. Table generated using pandas.to_latex() along with long table package

IX Code Appendix

```
import numpy as np
```

```
import struct
import cvxpy as cp
import os
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.dates as mdates
import matplotlib.lines as lines
import matplotlib.cbook as cbook
import cvxopt as cx
from cvxopt.modeling import op, dot, variable
from sklearn.linear_model import Ridge
from scipy.sparse.linalg import lsmr
from sklearn import linear_model
from decimal import Decimal

filePath = '/home/bdvr/Documents/GitHub/AMATH563/hw1/'
exploreIncorrect = False
debug = False

def read_idx(filename):
'''
reads in the binary input. Stolen from : https://gist.github.com/tylerneylon/ce60e8a06
'''
with open(filename, 'rb') as f:
zero, data_type, dims = struct.unpack('>HBB', f.read(4))
shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)

def loss_fn(X, Y, beta):
return cp.norm(cp.matmul(X, beta) - Y, 2)**2

def loss_fn_l1(X, Y, beta):
return cp.norm1(cp.matmul(X, beta) - Y)

def regularizer(beta):
'''
not used, was used to explore cvxpy
'''
return cp.pnorm(beta, p=2)**2

def objective_fn(X, Y, beta, lambd,oneOrTwo):
'''
not used, was used to explore cvxpy
'''
```

```
if(oneOrTwo==2):
return loss_fn(X, Y, beta) + lambd * regularizer(beta)
else:
return loss_fn_l1(X, Y, beta) + lambd * regularizer(beta)

def mse(X, Y, beta):
return (1.0 / X.shape[0]) * loss_fn(X, Y, beta).value
#the above four functions have been stolen from here https://www.cvxpy.org/examples/ma
def cvxExample(m,n):
'''
not used, was used to explore cvxpy
'''
np.random.seed(1)
A = np.random.randn(m, n)
b = np.random.randn(m)
x = cp.Variable(n)
#objective = cp.Minimize(cp.sum_squares(A*x - b))
objective = cp.Minimize(objective_fn(A,b,x,1.1,1))

#constraints = [0 <= x, x <= 1]
prob = cp.Problem(objective)

print(A)
print("Optimal value", prob.solve())
print("Optimal var")
print('MSE: '+str(mse(A,b,x.value)))
return(x.value)

def plotGridStyle(data,type,number,filename = None,save=False):
'''
plots and saves the weight vector for a given model number and penalty
'''
plt.clf()
plt.title(type+' ' + str(number) + ' weights')
plt.imshow(data,aspect='auto', cmap='gray')
plt.colorbar()
if(save):
plt.savefig(filename)
else:
plt.show()

def reshapeSols(sol,penalty,type,filename = None,save=False):
'''
part of the model fitting. Calls the plotGridStyle function and reshapes a 784x1 to 28
```

```
'''
counter = 0
getcol = np.array(sol[0,:])
depth = len(getcol)
while(counter<depth):
if(debug):
print(np.sqrt(len(data)))#debugging purposes
data = np.array(sol[:,counter])
penalty = str(penalty).replace('.','_')
fileExt = filename + 'weights_p'+str(penalty)+'_' + str(counter)
temp = np.reshape(data,(28,28))
plotGridStyle(temp,type,counter,fileExt,save)
counter+=1

def getThisNumber(train,labels,number):
'''
reshapes output arrays for binary classification task
'''
counter = 0
size = len(labels)
toFilter = np.zeros((size,1))
while(counter < size):
if(labels[counter,number] == 1 ):#or labels[counter,number+1] == 1
toFilter[counter,0]=1
counter+=1
return toFilter #https://stackoverflow.com/questions/44142173/how-can-a-numpy-array-of

def simpleSolutionAXB(train,labels,oneOrTwo,penalty):
'''
not used
'''
np.random.seed(1)
rows = train.shape[0]
cols = train.shape[1]
shape = (784,10)
print(shape)
x=cp.Variable(shape)
objective = cp.Minimize(objective_fn(train,labels,x,penalty,oneOrTwo))
prob = cp.Problem(objective)

#print(A)
print("Optimal value", prob.solve(verbose=True))
#print('MSE: '+str(mse(train,labels,x.value)))
return(x.value)
```

```
def cvxoptAttempt(train,labels):
'''
defunct
'''
A = cx.matrix(train)
B = cx.matrix(labels)
x = variable()
holdsol = op(objective_fn(A,B,x,0,2))
sol = holdsol.solve()
#print(sol['x'])
def reshapeLabels(numbers):
'''
reshapes output when first reading in the data
'''
numEntries = len(numbers)
result = np.zeros([10,numEntries])
i=0
while i < numEntries:
result[numbers[i],i]=1
i+=1
return result

def reshapeLeastSquareRes(data):
'''
unused, was used to gauge accuracy at one point
'''
res = np.ones([784,10])
counter = 0
for i in data:
if(np.count_nonzero(i)==0):
print(counter)
res[counter][:] = i
counter+=1
return res

def saveData(data,filename):
'''
saves numpy array, usually a weight vector
'''
np.save(filename,data)

def calcError(A,b,x,ord=None,axis=None):
'''
```

```
used to compute MSE
'''
if(axis is None):
return np.linalg.norm(b-A.dot(x),ord=ord) #https://docs.scipy.org/doc/numpy/reference/
else:
return np.linalg.norm(b-A.dot(x),ord=ord,axis=axis)

def linAlgSol(A,b):
'''
used to calculate least squares solution
'''
x = np.linalg.lstsq(A,b)[0] #https://docs.scipy.org/doc/numpy-1.13.0/reference/generat
return x

def pInvSol(A,b):
'''
unused
solves least squares using pseudo inverse
'''
pinv = (np.linalg.pinv(A)) #https://docs.scipy.org/doc/numpy/reference/generated/numpy
x= pinv.dot(b)
return x

def regLstSqr(A,b,penalty):
'''
sklearns ridge regression Regularized Least Squares function used to solve our model.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html
minimizes: ||y - Xw||^2_2 + alpha * ||w||^2_2
'''
clf=Ridge(alpha=penalty,solver='svd')
clf.fit(A,b)
return clf.coef_

def lasso(A,b,penalty):
'''
sklearns lasso regression function used to solve our model.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html
minimizes: (1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
note the averaging term of 1/2*n. Influences our penalty
'''
clf = linear_model.Lasso(alpha=penalty)
clf.fit(A,b)
return (clf.coef_)
```

```
def createBar(sol,size,title,save,fileLoc,penalty,type,ext=''):
'''
plots the magnitude of weights.
'''
strpenalty = str(penalty).replace('.','_')

if(type == 'lasso'):
imageLoc = fileLoc+'images/lasso/'+ext+'/'
else:
imageLoc = fileLoc+'images/RLS/'+ext+'/'

if(ext):
fileExt = imageLoc+ 'bar_p'+strpenalty+'_' + ext

xs = np.arange(784)
for i in np.arange(size):
plt.clf()
if(not ext):
newTitle = title + ' on ' + str(i)
fileExt = imageLoc+ 'bar_p'+strpenalty+'_' + str(i)
else:
newTitle = title
plt.title(newTitle)
plt.bar(x=xs,height=sol[i*784:((i+1)*784)])
plt.ylabel('weight value')
plt.xlabel('pixel index')
plt.axhline(0, color='red', lw=1)
if(not save):
plt.show()
else:
plt.savefig(fileExt)

def averageClassificationError(input,output,x,isMult = False,num=0):
'''
implements algorithm 1 highlighted in the paper for the binary classification task.
'''
res = input.dot(x)
trueSize=0
counter = 0
correct=0
size = len(input)
while(counter<size):
guess = round(res[counter][0])
if(guess==(output[counter])):
```

```
correct+=1
counter+=1
return correct/size

def batchClassificationError(input,output,x):
'''
Implements algorithm one for the batch trained case
'''
res = input.dot(x)
size = len(output)
correct = 0
counter = 0
classifications = np.zeros(10)
percentCorrect = np.zeros(10)
actuals = np.sum(output,axis=0)
while(counter<size):
guess = np.argmax(res[counter])
if(output[counter,guess]==1):
percentCorrect[guess]+=(1/actuals[guess])
counter+=1
return percentCorrect


def createDirectories(path):
'''
created directories to store results
'''
try:
os.mkdir(path)
except OSError:
if(debug):
print ("Creation of the directory %s failed" % path)
else:
if(debug):
print ("Successfully created the directory %s " % path)
#return lsmr(A=A,b=b,damp=penalty)[0]https://scikit-learn.org/stable/modules/generated

def trainData(smallDataIn,smallDataOut,penaltyRLS=10000,penaltyLasso=0.001,recompData=
'''
The meat of the code. Takes in a penalty for both RLS and Lasso along with the Data to
create images and store them for future reference.
'''
if(ext):
RLSFileName = fileLoc+'data/'+ext+'/RLSData_'+str(penaltyRLS)
```

```
LassoFileName = fileLoc+'data/'+ext+'/LassoData_'+str(penaltyLasso)
lassoImageLoc = fileLoc+'images/lasso/'+ext+'/'
RLSImageLoc = fileLoc+'images/RLS/'+ext+'/'
else:
RLSFileName = fileLoc+'data/RLSData_'+str(penaltyRLS)
LassoFileName = fileLoc+'data/LassoData_'+str(penaltyLasso)
lassoImageLoc = fileLoc+'images/lasso/'
RLSImageLoc = fileLoc+'images/RLS/'
size=len(smallDataIn)
if(recompData is True):
xRLS = regLstSqr(smallDataIn,smallDataOut,penaltyRLS)
lassoRes = lasso(smallDataIn,smallDataOut,penaltyLasso)
saveData(xRLS,RLSFileName)
saveData(lassoRes,LassoFileName)
else:
xRLS = np.load(RLSFileName+'.npy')
lassoRes = np.load(LassoFileName+'.npy')

if(ext):
xRLS = np.reshape(xRLS.T,(784,1))
lassoRes = np.reshape(lassoRes.T,(784,1))
else:
xRLS = np.reshape(xRLS.T,(784,10))
lassoRes = np.reshape(lassoRes.T,(784,10))

RLSErr = calcError(smallDataIn,smallDataOut,xRLS,2)
LErr = calcError(smallDataIn,smallDataOut,lassoRes,2)
#plz = np.sum(xRLS.T-xlin) #was for debugging
#print('cmon' + str(plz))


reshapeSols(lassoRes,penaltyLasso,'lasso',lassoImageLoc,savefile)
reshapeSols(xRLS,penaltyRLS,'RLS',RLSImageLoc,savefile)
if(debug):
print('lasso')
printStats(lassoRes,LErr,size)
print('reg lst sqr')
printStats(xRLS,RLSErr,size)
if(exploreIncorrect is True):
xlin = linAlgSol(trainIn,trainOut)
xinv = pInvSol(trainIn,trainOut)
print('pinv')
reshapeSols(xinv,0,'pinv','',False)
print('lstSqr')
```

```
reshapeSols(xlin,0,'lstSqr','',False)
return [lassoRes,xRLS]

def printStats(xs,err,size):
'''
used to debug the model weights
'''
print('Size:' + str(size))
print('Mean: ' + str(np.mean(xs)))
print('StdDev: ' + str(np.std(xs)))
print('Max: ' + str(np.max(xs)))
print('Min: ' + str(np.min(xs)))
print('Total Squared Error: ' + str(err))
print('Mean Squared Error: ' + str(err/size))

def debugData(xs,ys):
'''
used to debug data values
'''
print('input')
print(xs.shape)
print('Mean: ' + str(np.mean(xs)))
print('StdDev: ' + str(np.std(xs)))
print('Max: ' + str(np.max(xs)))
print('Min: ' + str(np.min(xs)))
print('output')
print(ys.shape)
print('Mean: ' + str(np.mean(ys)))
print('StdDev: ' + str(np.std(ys)))
print('Max: ' + str(np.max(ys)))
print('Min: ' + str(np.min(ys)))

def shapeData(rawIn,rawOut,smallSize=100):
'''
used to format the binary data after reading it in
'''
counter = 0
location=0
numRows = len(rawIn)
modTrain = np.zeros([numRows,28*28])
newY = reshapeLabels(rawOut)
modInSmall =  np.zeros([smallSize,28*28])
modOutSmall =np.zeros([10,smallSize])
for x in rawIn:
```

```
modTrain[counter] = rawIn[counter].flatten()/255
if(numRows-smallSize<counter):
modInSmall[location] = rawIn[counter].flatten()/255
modOutSmall[:,location] = newY[:,counter]
location+=1
counter+=1
newYT = newY.T
modOutSmall = modOutSmall.T
return modTrain,newYT,modInSmall,modOutSmall

def writeLine(filename,line):
'''
used to write our results
'''
with open(filename,'a') as fd:
fd.write(line) #https://stackoverflow.com/questions/2363731/append-new-row-to-old-csv-

def formRow(type,num,penalty,trainCE,trainMSE,testCE,testMSE,batch):
'''
formats a string to be written into our result file
'''
finalString = str(num) +','+str(trainMSE)+','+str(trainCE)+','+str(testMSE)+','+str(te
return finalString

def gatherTopXData(weights):
'''
implements algorithm 2 highlighted in the paper to select the most important pixels
'''
counter = 0
absWeights = np.absolute(weights.flatten())
size = len(weights)
potentialSol = np.argsort(absWeights)[::-1]
total = np.sum(absWeights)
pixels=[]
threshhold = 0.9
totalCoverage=0.0
newX = np.zeros([size,1])
while(counter<size and totalCoverage<threshhold):
curIndex = potentialSol[counter]
curVal= weights[curIndex]
counter+=1
newX[curIndex]=curVal
totalCoverage+=np.abs(curVal/total)
return newX
```

```
def createNewSparseFullModel(weights,numbers):
'''
handles the gatherTopXData for the batch trained case
'''
fullModel = np.zeros([784,10])
for n in numbers:
sparseWeights = np.reshape(gatherTopXData(weights[:,n]),(784,1))
fullModel[:,n] = sparseWeights.flatten()
return fullModel


'''
the rest below runs the scripts and calls appropriate functions to calculate results.
If you want to run this code on your own machine, you will have to set recompData to t
'''
#np.fromfile('/home/bdvr/Documents/GitHub/AMATH563/hw1/data/t10k-images-idx3-ubyte',)
types = ['lasso','RLS']
numbers = [0,1,2,3,4,5,6,7,8,9]

trainInputRaw = read_idx(filePath+'data/train-images-idx3-ubyte')
trainOutputRaw = read_idx(filePath+'data/train-labels-idx1-ubyte')

testInputRaw = read_idx(filePath+'data/t10k-images-idx3-ubyte')
testOutputRaw = read_idx(filePath+'data/t10k-labels-idx1-ubyte')


trainIn,trainOut,smallTrainIn,smallTrainOut = shapeData(trainInputRaw,trainOutputRaw,1
testIn,testOut,smallTestIn,smallTestOut = shapeData(testInputRaw,testOutputRaw,200)

debugSum = 0
runIndiv = True
counter=0
write0 = False
recompData = False
writeToCSV = False
writeToSparse = False
indivPenalties = {'lasso':0.001,'RLS':10000}
fullPenalties = {'lasso':0.01,'RLS':1000.0}
trainSize = len(trainIn)
testSize = len(testIn)
csvFile = filePath+'results/fullresults0.csv'
sparseCsvFile = filePath+'results/sparseResults0.csv'
```

```
if(runIndiv):
for x in numbers:
counter=0
createDirectories(filePath+'/data/'+str(x))
createDirectories(filePath+'/images/lasso/'+ str(x))
createDirectories(filePath+'/images/RLS/'+str(x))
numts = getThisNumber(trainIn,trainOut,x)
singleTest = getThisNumber(testIn,testOut,x)
#plotGridStyle(np.reshape(numtr[5000,:],(-1,28)),'whatev',0,False)
models = trainData(trainIn,numts,indivPenalties['RLS'],indivPenalties['lasso'],recompD
for y in models:
sparseWeights = gatherTopXData(y)
title = 'pixel weights for ' + str(x) + ' using ' + types[counter]
createBar(y.flatten(),y.shape[1],title,True,filePath,indivPenalties[types[counter]],ty

t = (types[counter])
num = (x)
TRCE = (1-averageClassificationError(trainIn,trainOut[:,num],y))
TRMSE = (calcError(trainIn,np.reshape(trainOut[:,num],(-1,1)),y,2)/trainSize)
TSCE = (1-averageClassificationError(testIn,testOut[:,num],y))
TSMSE = (calcError(testIn,np.reshape(testOut[:,num],(-1,1)),y,2)/testSize)

sparseTRCE = (1-averageClassificationError(trainIn,trainOut[:,num],sparseWeights))
sparseTRMSE = (calcError(trainIn,np.reshape(trainOut[:,num],(-1,1)),sparseWeights,2)/t
sparseTSCE = (1-averageClassificationError(testIn,testOut[:,num],sparseWeights))
sparseTSMSE = (calcError(testIn,np.reshape(testOut[:,num],(-1,1)),sparseWeights,2)/tes
if(writeToSparse):
insert = formRow(t,num,indivPenalties[types[counter]],sparseTRCE,sparseTRMSE,sparseTSC
if(write0 is True):
if(types[counter]=='RLS'):
writeLine(sparseCsvFile,insert)
else:
writeLine(sparseCsvFile,insert)

else:
print(t)
print(num)
print('sparse training classification error: ' + str(sparseTRCE))
print('sparse training MSE: ' + str(sparseTRMSE))
print('sparse testing classification error: ' + str(sparseTSCE))
print('sparse testing MSE: ' + str(sparseTSMSE))
if(writeToCSV):
insert = formRow(t,num,indivPenalties[types[counter]],TRCE,TRMSE,TSCE,TSMSE,False)
if(write0 is True):
```

```
if(types[counter]=='RLS'):
writeLine(csvFile,insert)
else:
writeLine(csvFile,insert)
else:
print(t)
print(num)
print('training classification error: ' + str(TRCE))
print('training MSE: ' + str(TRMSE))
print('testing classification error: ' + str(TSCE))
print('testing MSE: ' + str(TSMSE))

counter+=1
if(debug):
print(debugSum)

if(debug):
debugData(trainIn,trainOut)
models = trainData(trainIn,trainOut,fullPenalties['RLS'],fullPenalties['lasso'],recompl
counter = 0

for x in models:
title = 'pixel weights using ' + types[counter]
createBar(x.flatten(),x.shape[1],title,True,filePath,fullPenalties[types[counter]],typ
sparseFull = createNewSparseFullModel(x,numbers)
allSparseResTrain = batchClassificationError(trainIn,trainOut,sparseFull)
allFullResTrain = batchClassificationError(trainIn,trainOut,x)
allSparseResTest = batchClassificationError(testIn,testOut,sparseFull)
allFullResTest = batchClassificationError(testIn,testOut,x)
allSparseMSETrain = calcError(trainIn,trainOut,sparseFull,axis=0)
allSparseMSETest = calcError(testIn,testOut,sparseFull,axis=0)
allFullMSETrain = calcError(trainIn,trainOut,x,axis=0)
allFullMSETest = calcError(testIn,testOut,x,axis=0)
for y in numbers:
t = (types[counter])
num = y

TRCE = (1-allFullResTrain[num])
TRMSE = allFullMSETrain[numight be useful https://glowingpython.blogspot.com/2012/03/s
TSCE = (1-allFullResTest[num])
TSMSE = allFullMSETest[num]/testSize

sparseTRCE =  (1-allSparseResTrain[num])
sparseTRMSE = allSparseMSETrain[num]/trainSize
```

```
sparseTSCE =  (1-allSparseResTest[num])
sparseTSMSE =  allSparseMSETest[num]/testSize
if(writeToSparse):
insert = formRow(t,num,fullPenalties[types[counter]],sparseTRCE,sparseTRMSE,sparseTSCE
if(write0 is True):
if(types[counter]=='RLS'):
writeLine(sparseCsvFile,insert)
else:
writeLine(sparseCsvFile,insert)
else:
print(t)
print(num)
print('sparse training classification error: ' + str(sparseTRCE))
print('sparse training MSE: ' + str(sparseTRMSE))
print('sparse testing classification error: ' + str(sparseTSCE))
print('sparse testing MSE: ' + str(sparseTSMSE))
if(writeToCSV):
insert = formRow(t,num,fullPenalties[types[counter]],TRCE,TRMSE,TSCE,TSMSE,True)
if(write0 is True):
if(types[counter]=='RLS'):
writeLine(csvFile,insert)
else:
writeLine(csvFile,insert)

else:
print(t)
print(num)
print('training classification error: ' + str(TRCE))
print('training MSE: ' + str(TRMSE))
print('testing classification error: ' + str(TSCE))
print('testing MSE: ' + str(TSMSE))


counter+=1

'''
useful links:
https://www.cvxpy.org/examples/machine_learning/ridge_regression.html
might be useful https://glowingpython.blogspot.com/2012/03/solving-overdetermined-syst
'''
```