AMATH 563 Homework 1

Daniel Burnham (https://github.com/burnhamdr)

April 13, 2020

Abstract

The work presented here is motivated by material covered in AMATH 563 Inferring Structure of Complex Systems regarding the applications of Regression, Model Selection. Regression analysis is a data driven mathematical modelling method to find a combination of model parameters that best explains the data. This definition requires special focus on how model parameters are combined (linear or non-linear) and on the metric by which an explanation, or model fit, is deemed the best. Linear regression is exclusively used for modelling the data in the work presented here, but various approaches to constrain the model coefficients are explored. This report will discuss these topics in the context of a data set of handwritten numerical digits. The data consists of image data for samples of ten different handwritten digits (0 through 9). Regression methods are applied to the pixel data for each digit and are compared and contrasted specifically referencing model complexity (i.e. the number of large model coefficients). Least absolute shrinkage and selection operator (LASSO) regression presents as especially useful in this work for its promotion of model term sparsity. Ridge regression and Elastic Net models combining both Ridge and Lasso regression characteristics are also investigated for model complexity reduction. Results from the work presented here will be grouped into four distinct sections. First various regression methods are employed and the largest model terms (i.e. largest model contributors) are compared. Secondly, sparsity promotion is pursued through investigating the Elastic Net method and manipulating model fit parameters associated. Thirdly, model performance is assessed on a hold out set of data to characterize the impact of model complexity on handwritten digit classification. Lastly, models are constructed to predict each digit individually and the results of these models are compared to those of the data modelled jointly for all digits. Overall, promoting model sparsity results in identifying key pixels for digit class discrimination but subsequently using these digits to reduce model complexity results in increasingly poor classification performance. Additionally, performing regression analysis on each digit individually is met with a severe class imbalance problem that even when rectified with minority class up-sampling significantly hampers the performance of regression models for many digit classes. Regression and model selection are important concepts underlying a significant amount of modern data science and machine learning. As such, the work presented here represents a description of theory and application in an effort to provide a foundation of understanding for future learning in data driven modelling.

1 Introduction and Overview

Over the last decade data has emerged as an important commodity in many industries. The value of data however is rooted in the collective information that is represented by large swaths of data points. Identifying and extracting this information from data sets is thus of great importance to realizing the potential of collected data for scientific discovery, engineering applications, and general data driven decision making. Machine learning has grown to be colloquially synonymous with the extraction of information from data, and though in many ways this association is appropriate, the umbrella that is the modern term "machine learning" tends to mask the underlying age-old mathematical principles. In many ways this represents progress as drawing information from data has never been more accessible, but in other ways the ubiquity of ready-made machine learning methods deters fundamental understanding of model selection and implementation. In light of this, it is important to ground any exploration into data science methods in the unifying concept of regression as it is a seminal data modelling method upon which many more sophisticated approaches build.

The work presented here explores the application of regression, specifically linear regression, in the context of classifying image data of hand written numerical digits into appropriate digit classes. This problem is of particular interest in data science as translating hand written information into digital typeset is of practical importance in many information systems. The data set utilized here is the MNIST database (http://yann.lecun.com/exdb/mnist/) of handwritten digits which consists of a training set of 60,000 examples, and a test set of 10,000 examples. The training set was first used for finding a map from the image space to the label space using various $AX = B$ solver methods where the matrix A contains the flattened pixel data for each hand written digit image, X represents the coefficients mapping pixel values to label space, and B represents the target label space values with each column representing a digit class and each row corresponding to the respective row entry in A.

It is evident that this data set forms an overdetermined system of equations for determining an $AX = B$ mapping, i.e. there are more equations than unknowns. This indicates that the results of individual equations will be inconsistent in general thus representing the need for constraints to be imposed on the type of solution arrived at for X. A common approach to determining X is a least square criterion where instead of demanding that $AX = B$, we seek an X that minimizes the average squared difference between AX and B. In other words, we are looking for a mapping X that minimizes the sum-squared error: $\|AX - B\|_2^2$. This approach serves to find the best possible solution as measured by the 2-norm. Other approaches are available however that define what ”best possible” looks like in terms of other minimization constraints (see Section 2.1). The methods employed here for determining the mapping from image to label space are: the Moore-Penrose pseudo-inverse, least squares, Lasso regression, Huber Regression, and Ridge regression. These will be described in further detail in Section 2.

Once a mapping $AX = B$ has been determined, of particular interest in many instances are which data set features contribute the most to this mapping (i.e. have the largest corresponding coefficients in X). This can be determined by examining the magnitude of the values of X termed model coefficients or loadings. These values correspond to specific features of A and the larger the loading of a feature, the more it contributes to how the data set is mapped to B thus indicating relative feature significance. To best accomplish feature selection it is therefore preferable that the method employed for constraining the solution X is one that stratifies the loadings. In other words, a method that constrains the chosen X to have few large values and many values close to zero will best highlight significant data set features. In practice this is most often best accomplished through the implementation of least absolute shrinkage and selection operator (LASSO) which is an ℓ_1 penalized regression technique (Brunton 2019). LASSO is used here to examine which pixels most contribute to hand written digit classification. The most significant pixels are compared across mapping methods listed earlier, and also specifically in implementations of LASSO regression with varying model fit parameters. Lastly, when mappings are determined for individual pixels the most significant pixels for classifying each digit class are identified.

2 Theoretical Background

2.1 Linear Regression

Much of modern machine learning and data science involves solving an optimization problem. A fundamental example of this is regression for data modelling. Regression models arise as solutions to the system of equations $AX = B$ where the values of A represent the input data and B represents the output values that the data corresponds to. The matrix X therefore represents the mapping of the data in A to the outputs in B. In data science it is often the case that these regression problems are either massively over or underdetermined meaning that the data in A has either more instances (rows) than parameters (columns) or vice versa. In the case of an overdetermined system there are more equations than unknowns resulting in a system of equations that do not agree on a solution X. In the underdetermined case there are fewer equations than unknowns meaning that there are either zero solutions or infinitely many. In either case, a constraint or penalty is needed to settle on a single solution. This constraint is termed regularization and defines the regression optimization problem. The underlying optimization objective functions for finding a linear map in either an overdetermined (1) or underdetermined (2) case are given by (Brunton 2019):

$$\underset{X}{\arg\min}\ (\|AX - B\|_2 + \alpha g(x)) \tag{1}$$

$$\underset{X}{\arg\min}\ g(X) \text{ subject to } \|AX - B\|_2 \leq \epsilon \tag{2}$$

where $g(x)$ is a given penalization, with penalty parameter α, in the case of an overdetermined system (Equation 1). The term g(x) is of particular interest because it can be used to modulate the characteristic(s) of the solution X that is sought in solving the linear systems of equations $AX = B$ through optimization. The regularization term $g(x)$ can take on different forms depending on the type of regression:

$$\textbf{Ordinary Least Squares: } \arg\min_{X} (\|AX - B\|_2^2) \tag{3}$$

$$\textbf{Ridge: } \arg\min_{X} (\|AX - B\|_2^2 + \alpha \|X\|_2^2) \tag{4}$$

$$\textbf{Lasso: } \arg\min_{X} (\frac{1}{2n} \|AX - B\|_2^2 + \alpha \|X\|_1) \tag{5}$$

$$\textbf{Elastic Net: } \arg\min_{X} (\frac{1}{2n} \|AX - B\|_2^2 + (1-\beta)\frac{\alpha}{2} \|X\|_2^2 + \beta\alpha \|X\|_1) \tag{6}$$

$$\textbf{Huber (Robust): } \arg\min_{X,\sigma} (H_\epsilon(\frac{AX - B}{\sigma}\sigma) + \|X\|_2^2) \tag{7}$$

$$\text{where: } H_\epsilon(z) = \begin{cases} z^2 & if\,|z| < \epsilon, \\ 2\epsilon\,|z| - \epsilon^2 & \text{otherwise} \end{cases}$$

It is evident in each case (Equations 3-7) that the regularization is distinct. Ordinary least squares does not include a penalty at all. Ridge regression adds a penalty based on the ℓ_2 of X. Lasso regression penalizes the solution based on the ℓ_1 norm. Elastic net combines both Ridge and Lasso penalties in proportions defined by β. Huber, or robust, is most similar to Ridge regression in that the solution is penalized based on the ℓ_2 norm, but differs in that it minimizes a loss function that is linear for samples that are classified as outliers. A sample is classified as an inlier if the absolute error of that sample is less than a certain threshold (Huber 2009). This helps ensure that the loss function is not heavily influenced by the outliers while not completely ignoring their effect.

2.2 Performance Metrics

In assessing the performance of the regression models, confusion matrices will be used. A confusion matrix for a classification problem simply indicates the frequency of correct and erroneous class predictions in the test data. The counts indicate how often a class is predicted as itself or if it is frequently misclassified as another class. This is important information in assessing the performance and thus suitability of the classification model. In addition, Accuracy, Precision, Recall, and F1 metrics will be reported on. These metrics depend on the frequencies of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) classification cases. Taken together these give a more holistic view of classification performance.

$$\textbf{Accuracy: } \frac{TP + TN}{TP + TN + FP + FN}$$

$$\textbf{Precision: } \frac{TP}{TP + FP}$$

$$\textbf{Recall: } \frac{TP}{TP + FN}$$

$$\textbf{F1: } 2\frac{precision * recall}{precision + recall}$$

Accuracy alone does not offer insight into the number predictions that are falsely classified positive and falsely classified negative. As such precision and recall supplement the accuracy information by determining the correctly classified cases out of all predictions for a given class (precision) and the correctly classified cases out of all classification instances of a given class (recall). The F1 score gives more weight to false negatives and false positives while not letting large numbers of true negatives influence the score in an attempt to strike a balance between precision and recall. F1 is a particularly good metric when class imbalance is present in the data used for a classification task.

3 Algorithm Implementation and Development

In the case of the image pixel data of interest here, the system of equations is overdetermined. The matrix A has 60000 rows (instances) and 784 columns (parameters) of pixel data. Each row represents a different flattened 28x28 handwritten numerical digit image. The matrix B therefore must contain 60000 instances as well, representing an output (classification) for each image contained in A. Additionally, since a given image in A is classified as a discrete digit type, B must have 10 columns each representing a digit class. Therefore each row in B appears as a "one-hot" encoded vector with a 1 in the column representing the digit class of the corresponding row in A. Taking the dimensions of A and B into account, it is evident that the mapping from image to label space encapsulated by X must have 784 rows and 10 columns such that each row in A is linearly transformed by each column in X to yield the "one-hot" encoded row vector representing the digit classification in B. As handwritten digits take on diverse forms, the pixel data for each instance of the same digit class in A will require the values of X to take on a form such that, regardless of handwritten form, the image is mapped to the correct class. This is where the optimization problem arises. A solution X to the linear system of equations $AX = B$ is desired such that that the two norm of a matrix containing the differences between the mapping AX and the actual class values B is minimized.

Overall, the main steps of the work described in this report are as follows:

1. Import MNIST image data train and test sets (http://yann.lecun.com/exdb/mnist/). Ensure that the test data remains held out of subsequent analysis.
2. Flatten each image into single array, concatenate all images together to form the data matrix A. Translate the single array of digit class values, ranging from 0 to 9, to a one-hot encoded matrix with each column representing a digit class and each row corresponding to an image instance in A.
3. Compute mapping from image space to label space using various $AX = B$ solvers that employ different regularizations (See Section 2.1). Employ K-fold cross validation of this mapping to demonstrate variability in model parameters and error.
4. Plot largest cross validated model coefficients.
5. Compute mapping again while using sparsity promoting regularization strategies. Again employ K-fold cross validation to demonstrate the variability of the loadings.
6. Compute mappings, with the selected sparsity promoting method from the previous step, to map the image data to each digit class individually. Employ K-fold cross validation as well as minority class up-sampling to address the impact of class imbalance on the fit model.

4 Computational Results

4.1 Implementing $AX = B$ Solvers

The computational results in this section are a result of computing mappings from image space to label space with different $AX = B$ solvers. The solvers used are described in detail in Section 2.1. The methods employed here are: the Moore-Penrose pseudo-inverse (uses least squares), least squares, Lasso regression, Huber Regression, and Ridge regression. The results for the Elastic Net solvers in Figure 1 used an l1_ratio (the β parameter discussed in Section 2.1) of 1 so as to behave as LASSO regressors. Two different alpha values are tested which modulate the strength of the Elastic Net ℓ_1 penalty.

4.2 Sparsity Promotion

In observing the top model parameters shown in Figure 1 the Elastic Net (LASSO) regressors appear to promote the most sparsity in model parameters (i.e. a few dominant large coefficients). Lasso regression, which implements a ℓ_1 norm penalty, is known to promote sparsity and, in comparing the Elastic Net (LASSO) results to that of Ridge regression, the power of ℓ_1 norm penalty to stratify model coefficients is

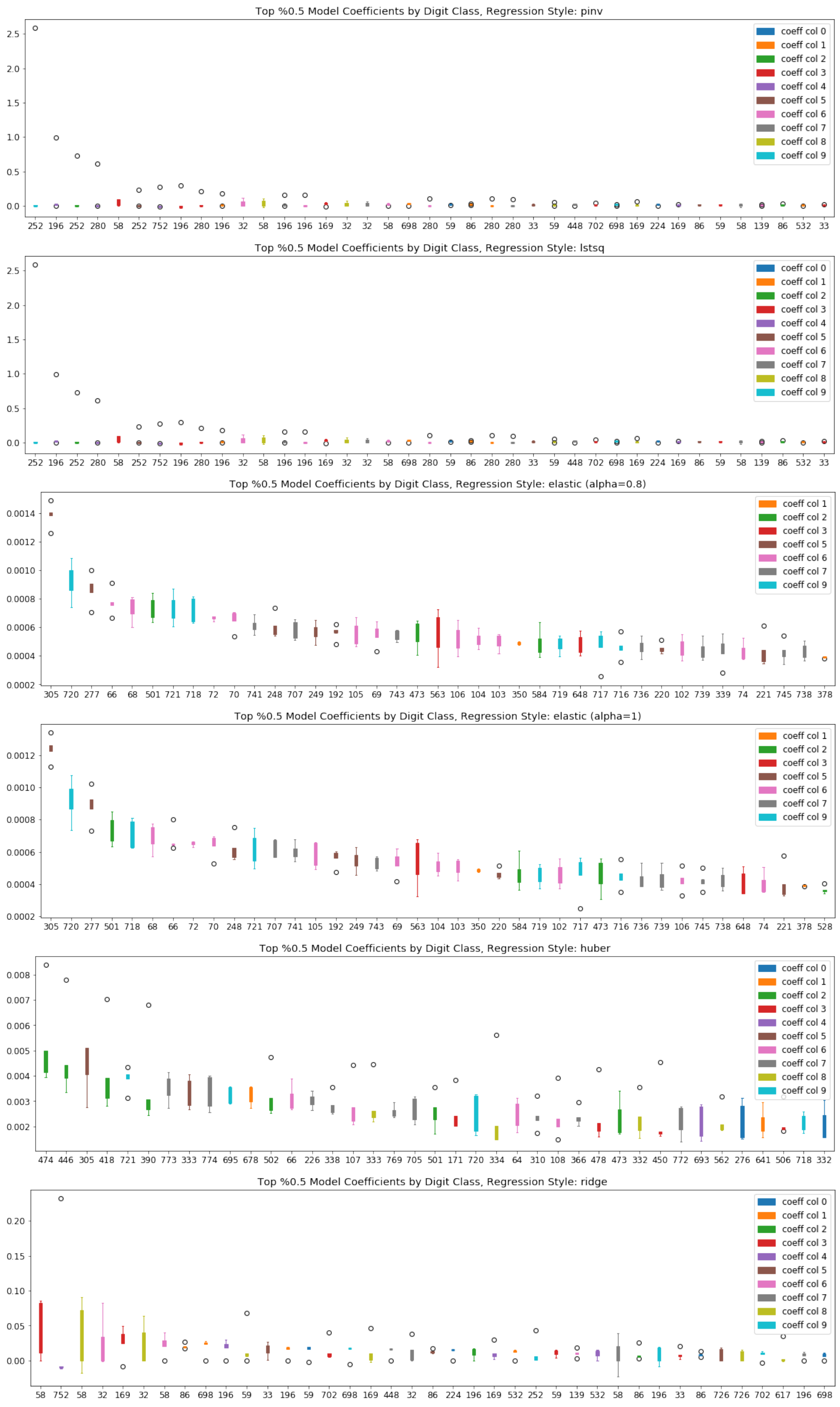


Figure 1: The top 39 coefficients for each $AX = B$ solver. X axis values indicate the pixel values multiplied by the coefficients in the mapping. The Y axis represents the size of the coefficient. The color scale indicates which column in X the coefficient value belongs to (i.e. to which digit class it maps).

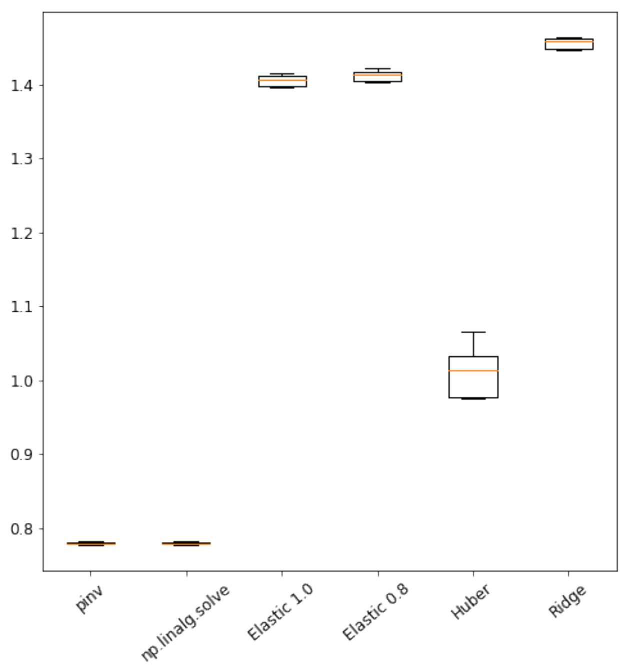


Figure 2: K-fold cross validated error between mapped B matrix and actual B matrix for each $AX = B$ solver. Y axis indicates the error value.

evident. As such, for the sparsity promotion investigation presented here, LASSO regression will be used (Elastic Net with l1_ratio of 1). The parameters β and α were modulated and tested to find the model promoting the most sparsity. The results of these experiments are illustrated in Appendix B. Also, model coefficient drop out was tested to observe the performance impact of sequentially adding dominant coefficients to the model. These results are presented in Appendix B. Here the model coefficients of the model with the most sparsity is presented (Figure 3) along with analysis of digit classification performance for this model (Figure 4).

4.3 Individual Digit Class Models

Lastly, the sparse model was then fit to each digit class individually to identify pixels most significant in the mappings for each digit (Figures 5 and 6). The performance for each individual model is shown in Appendix B.

5 Summary and Conclusions

The first significant result obtained was the demonstrated ability of ℓ_1 penalty regularization to yield sparse models. This served to highlight pixels (features) that contributed the most to the mapping and could be regarded as significant pixels involved in digit class discrimination. The second key result was that the classification performance of the ℓ_1 LASSO regression method was good. This implies there are indeed useful features that can discriminate a diverse set of handwritten numerical digits. However, when this sparse model was applied to each digit class individually this performance deteriorated (Figures 10-19). This may be due in large part to the severe class imbalance of this problem. When mapping to individual label space from image space there are far more cases of digits outside of the class of interest. Even with minority class upsampling the quality of model fit for each digit class is most likely reduced.

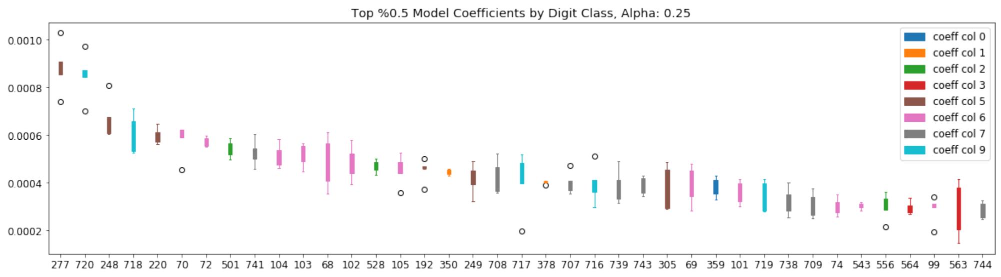
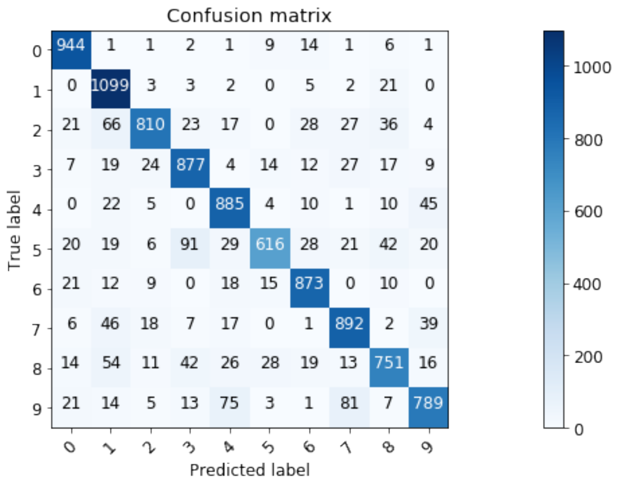
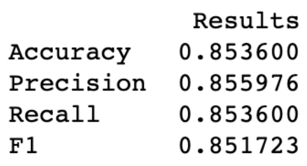


Figure 3: K-fold cross validated model coefficients of an Elastic Net model with $\alpha = 0.25$. X axis indicates the ranked order of pixel values multiplied by the coefficients in the mapping, thus illustrating the most important pixel features. The Y axis represents the size of the coefficient. The color scale indicates which column in X the coefficient value belongs to (i.e. to which digit class it maps).



	0	1	2	3	4	5	6	7	8	9
0	944	1	1	2	1	9	14	1	6	1
1	0	1099	3	3	2	0	5	2	21	0
2	21	66	810	23	17	0	28	27	36	4
3	7	19	24	877	4	14	12	27	17	9
4	0	22	5	0	885	4	10	1	10	45
5	20	19	6	91	29	616	28	21	42	20
6	21	12	9	0	18	15	873	0	10	0
7	6	46	18	7	17	0	1	892	2	39
8	14	54	11	42	26	28	19	13	751	16
9	21	14	5	13	75	3	1	81	7	789

(a)



	Results
Accuracy	0.853600
Precision	0.855976
Recall	0.853600
F1	0.851723

(b)

Figure 4: (a) Sparse model digit classification confusion matrix. (b) Classification performance metrics.

References

Brunton, Steven L. (Steven Lee) (2019). *Data-driven science and engineering : machine learning, dynamical systems, and control*. Cambridge, United Kingdom ; New York, NY: Cambridge University Press. ISBN: 9781108422093.

Huber, Peter J. (2009). *Robust statistics*. Second edition. Hoboken, N.J.: Wiley. ISBN: 9780470434680.

Appendix A Python Functions

- `numpy.linalg.pinv(a, rcond=1e-15, hermitian=False)`
  Compute the (Moore-Penrose) pseudo-inverse of a matrix. Calculate the generalized inverse of a matrix using its singular-value decomposition (SVD) and including all large singular values. The pseudo-inverse of a matrix A, denoted $A^{\dagger}$, is defined as: "the matrix that 'solves' [the least-squares problem] $Ax = b$.

- `numpy.linalg.lstsq(a, b, rcond='warn')`
  Return the least-squares solution to a linear matrix equation. Solves the equation $Ax = b$ by computing a vector x that minimizes the squared Euclidean 2-norm $\|b - Ax\|_2^2$. The equation may be under-, well-, or over-determined (i.e., the number of linearly independent rows of a can be less than, equal to, or

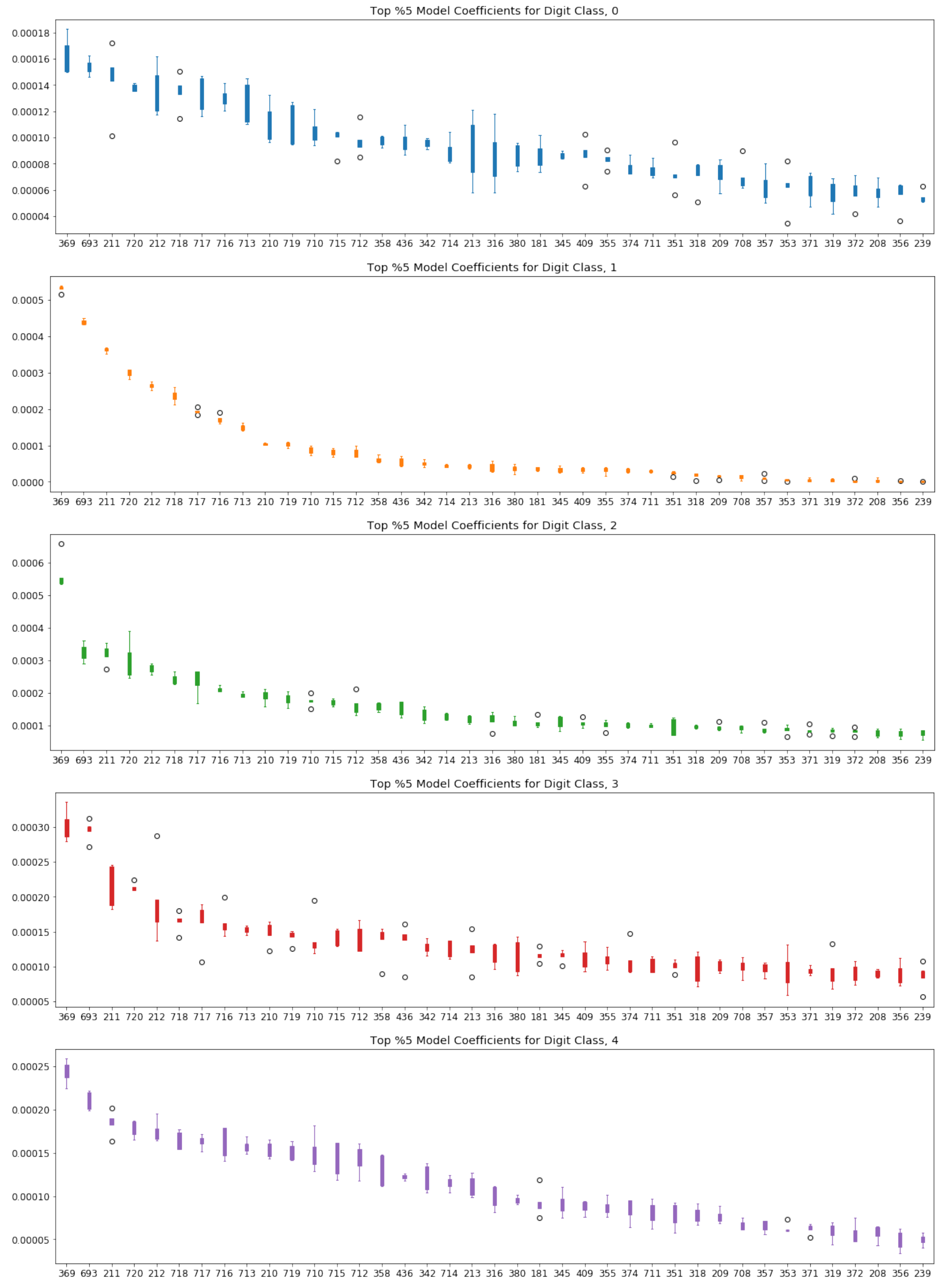


Figure 5: The top 39 coefficients for digit class specific $AX = B$ mapping for digits 0 through 4. X axis values indicate the pixel values multiplied by the coefficients in the mapping. The Y axis represents the size of the coefficient. The color scale indicates which column in X the coefficient value belongs to (i.e. to which digit class it maps).

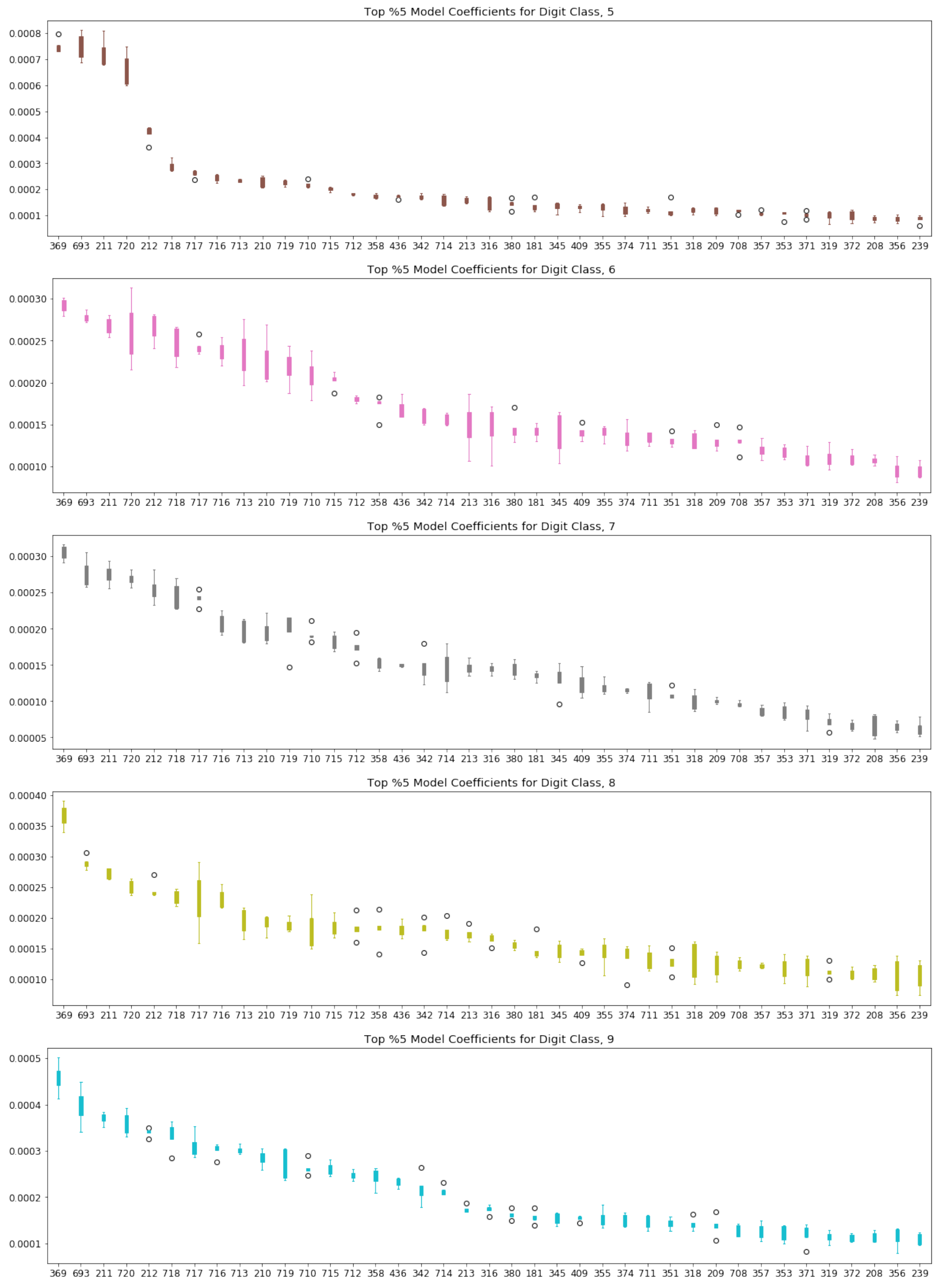


Figure 6: The top 39 coefficients for digit class specific $AX = B$ mapping for digits 5 through 9. X axis values indicate the pixel values multiplied by the coefficients in the mapping. The Y axis represents the size of the coefficient. The color scale indicates which column in X the coefficient value belongs to (i.e. to which digit class it maps).

greater than its number of linearly independent columns). If A is square and of full rank, then x (but for round-off error) is the "exact" solution of the equation.

- `sklearn.linear_model.Ridge(alpha=1.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, solver='auto', random_state=None)`
  Linear Model trained with L2 prior as regularizer (aka the Ridge).

- `sklearn.linear_model.Lasso(alpha=1.0, fit_intercept=True, normalize=False, precompute=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic')`
  Linear Model trained with L1 prior as regularizer (aka the Lasso).

- `sklearn.linear_model.ElasticNet(alpha=1.0, l1_ratio=0.5, fit_intercept=True, normalize=False, precompute=False, max_iter=1000, copy_X=True, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic')`
  Linear regression with combined L1 and L2 priors as regularizer.

- `sklearn.linear_model.HuberRegressor(epsilon=1.35, max_iter=100, alpha=0.0001, warm_start=False, fit_intercept=True, tol=1e-05)`
  Linear regression model that is robust to outliers. The parameter epsilon controls the number of samples that should be classified as outliers. The smaller the epsilon, the more robust it is to outliers.

Appendix B Supplemental Figures

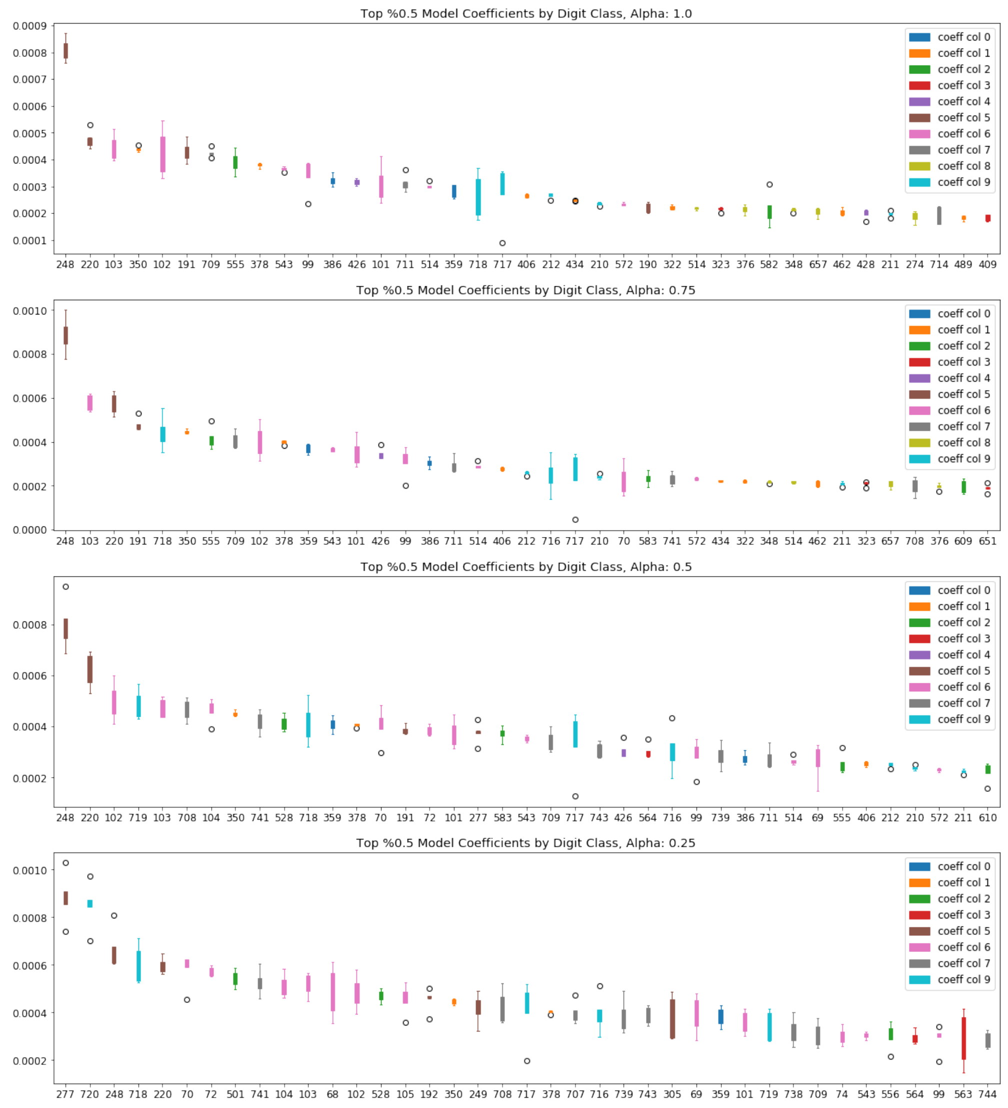


Figure 7: The top 39 coefficients for LASSO regression with constant l1 ratio of 1, but with varying α regularizer scaling. X axis values indicate the pixel values multiplied by the coefficients in the mapping. The Y axis represents the size of the coefficient. The color scale indicates which column in X the coefficient value belongs to (i.e. to which digit class it maps).

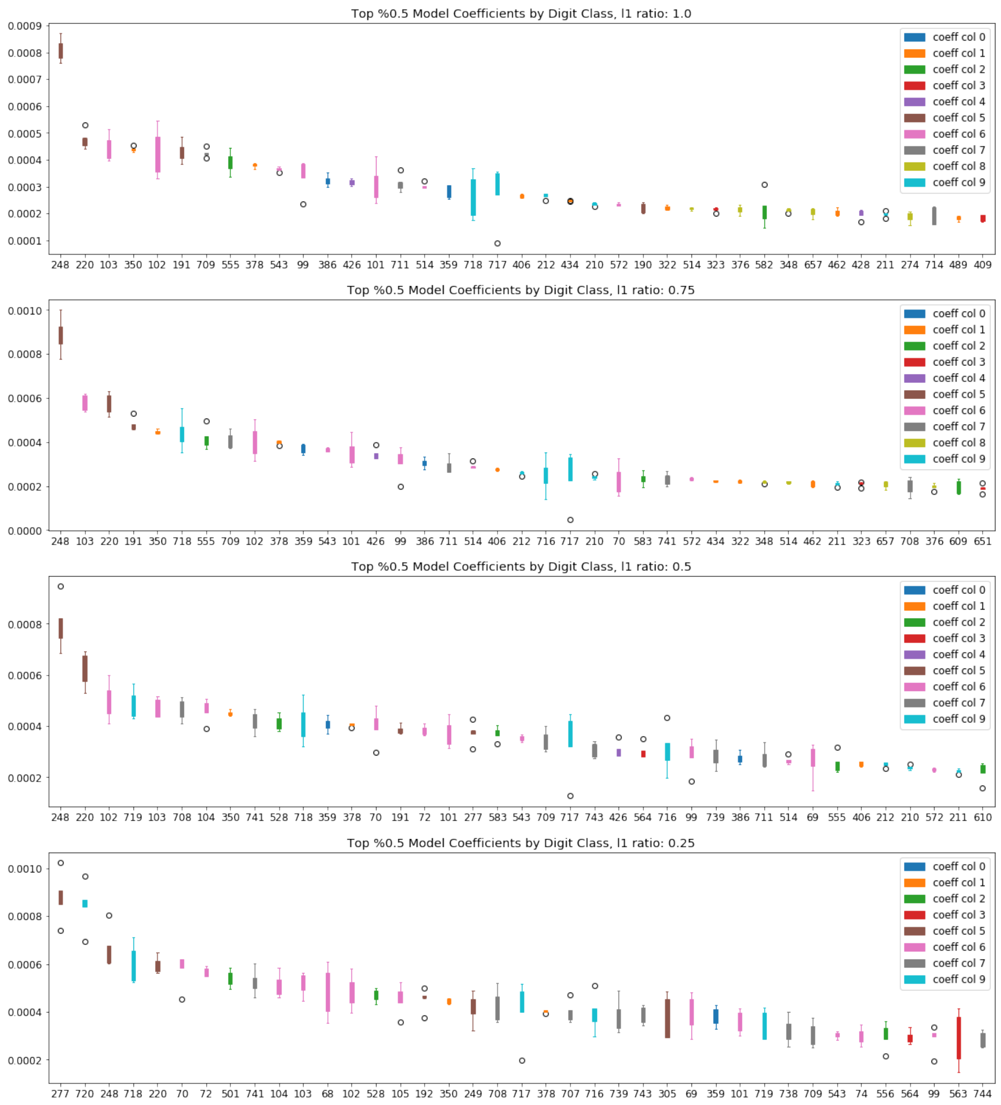


Figure 8: The top 39 coefficients for LASSO regression with constant α of 1, but with varying l1 ratios (i.e. ridge and lasso penalty mixing). A l1 ratio equal to 1 is pure lasso penalty. X axis values indicate the pixel values multiplied by the coefficients in the mapping. The Y axis represents the size of the coefficient. The color scale indicates which column in X the coefficient value belongs to (i.e. to which digit class it maps).

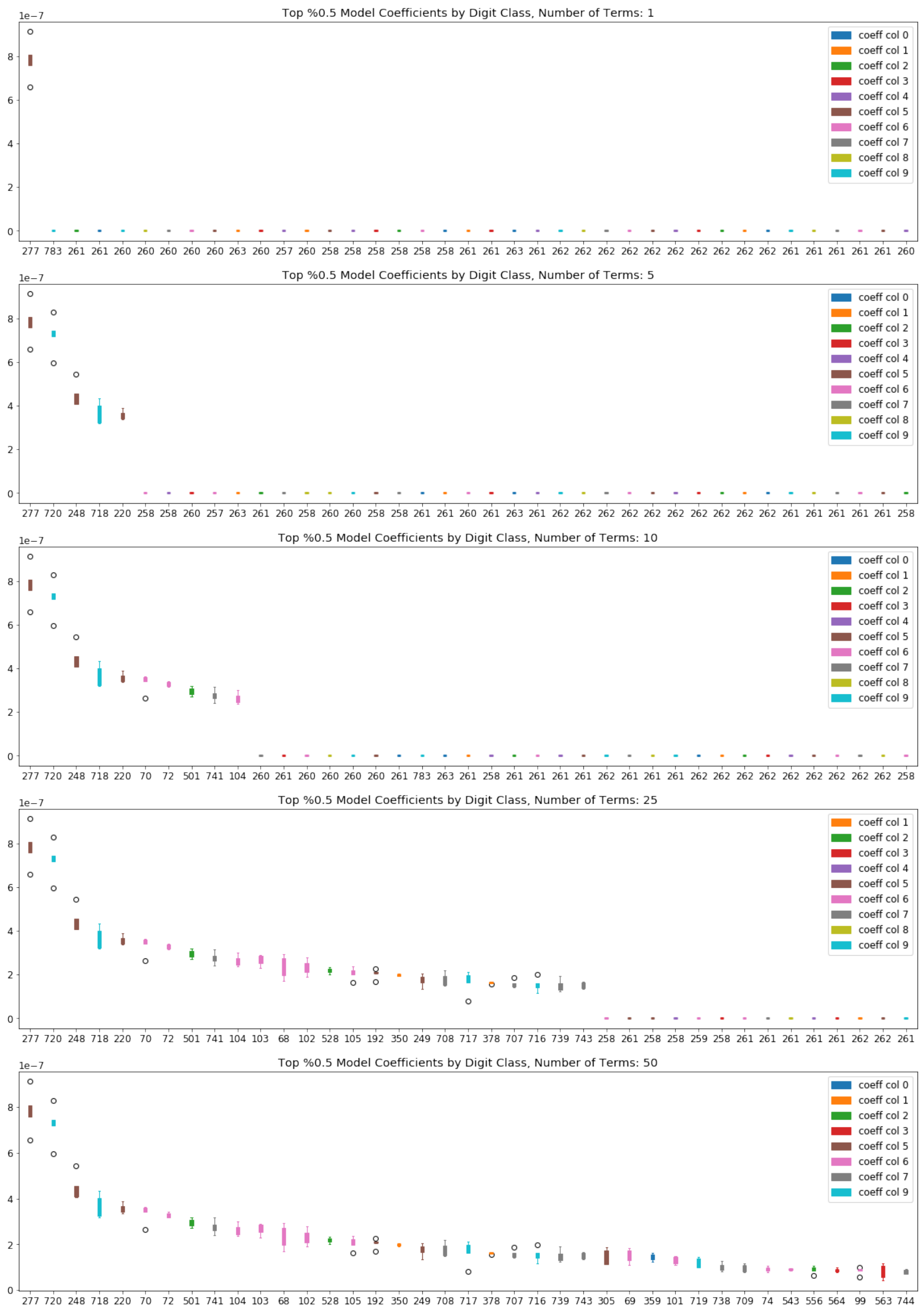
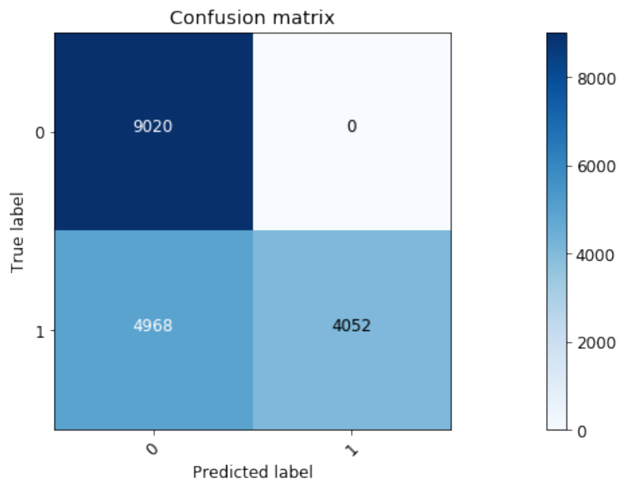
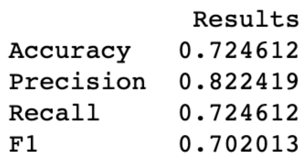


Figure 9: The top coefficients for the sparsity promoting LASSO $AX = B$ solver with an increasing number of included (non-zero) coefficients. This experiment continued to 150, 300, and 600 terms but the coefficients are not visualized here as the top 39 coefficients for these conditions are the same as seen in the case of 50 top coefficients.

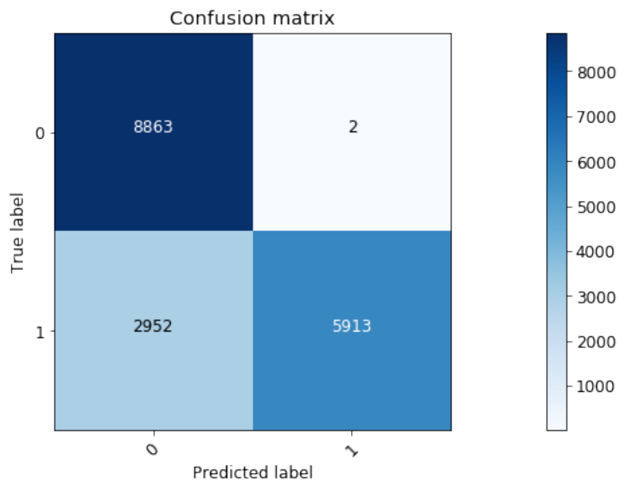


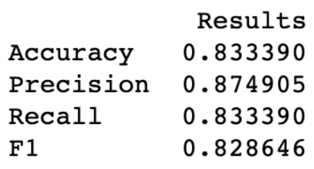


	Results
Accuracy	0.724612
Precision	0.822419
Recall	0.724612
F1	0.702013

(a) (b)

Figure 10: (a) Sparse model 0 digit specific classification confusion matrix. (b) Classification performance metrics.

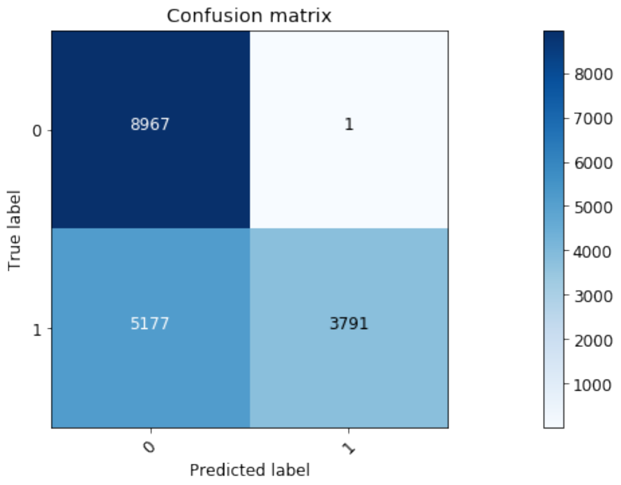


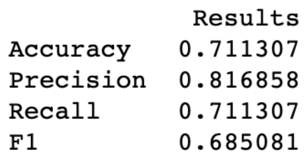


	Results
Accuracy	0.833390
Precision	0.874905
Recall	0.833390
F1	0.828646

(a) (b)

Figure 11: (a) Sparse model 1 digit specific classification confusion matrix. (b) Classification performance metrics.

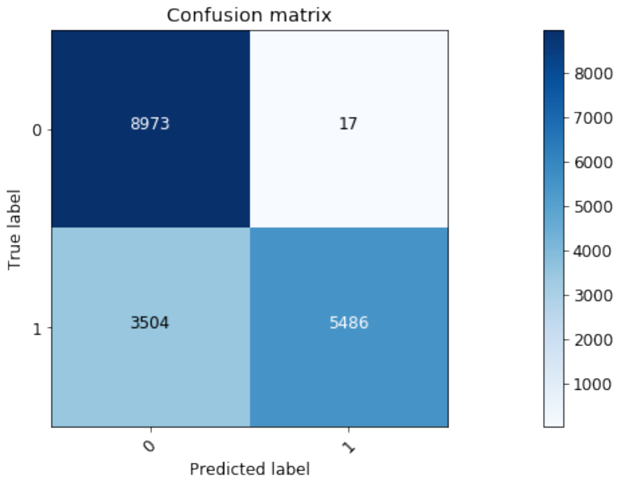


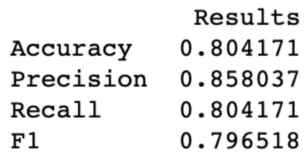


```
                Results
Accuracy       0.711307
Precision      0.816858
Recall         0.711307
F1             0.685081
```

(a) (b)

Figure 12: (a) Sparse model 2 digit specific classification confusion matrix. (b) Classification performance metrics.

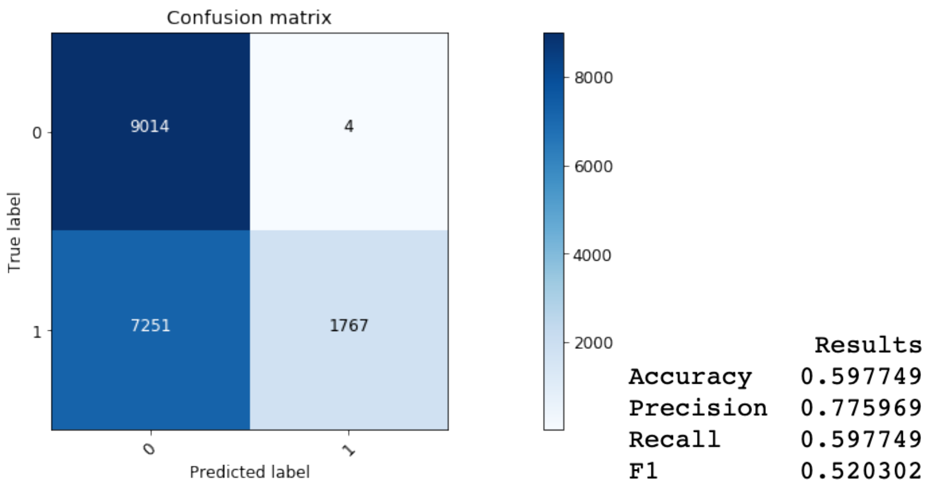




```
                Results
Accuracy       0.804171
Precision      0.858037
Recall         0.804171
F1             0.796518
```

(a) (b)

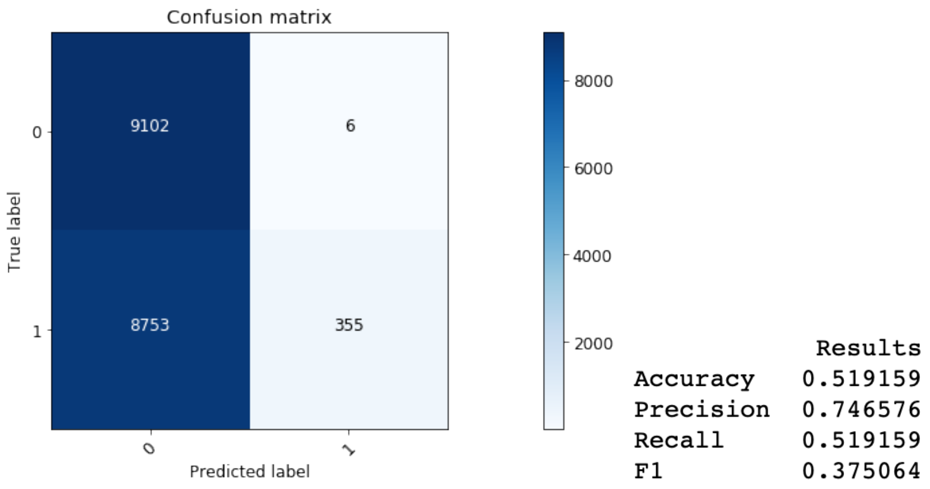
Figure 13: (a) Sparse model 3 digit specific classification confusion matrix. (b) Classification performance metrics.



	Results
Accuracy	0.597749
Precision	0.775969
Recall	0.597749
F1	0.520302

(a) (b)

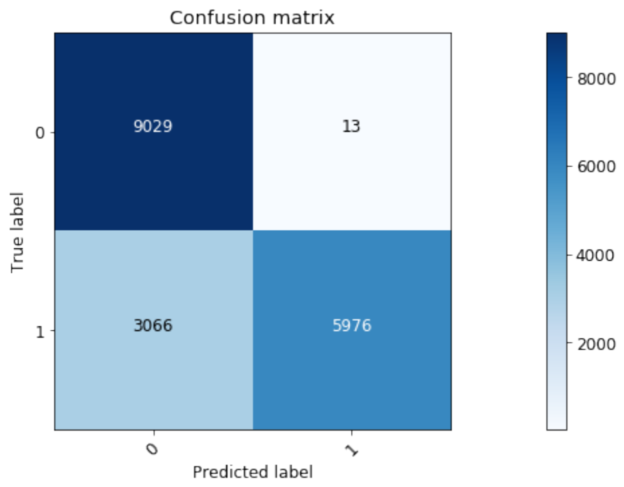
Figure 14: (a) Sparse model 4 digit specific classification confusion matrix. (b) Classification performance metrics.

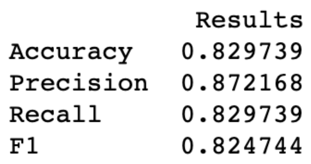


	Results
Accuracy	0.519159
Precision	0.746576
Recall	0.519159
F1	0.375064

(a) (b)

Figure 15: (a) Sparse model 5 digit specific classification confusion matrix. (b) Classification performance metrics.

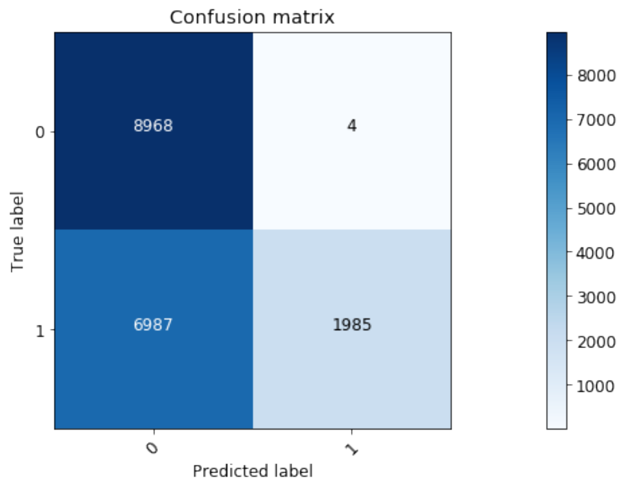


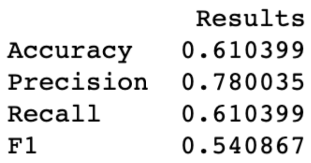


	Results
Accuracy	0.829739
Precision	0.872168
Recall	0.829739
F1	0.824744

(a) (b)

Figure 16: (a) Sparse model 6 digit specific classification confusion matrix. (b) Classification performance metrics.

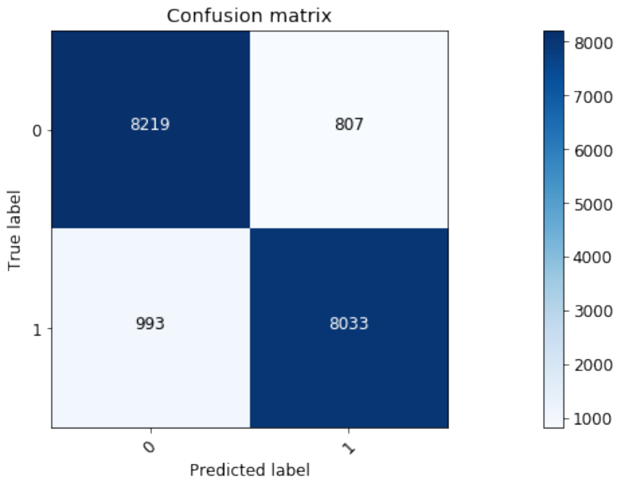




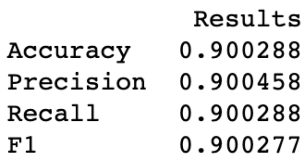
	Results
Accuracy	0.610399
Precision	0.780035
Recall	0.610399
F1	0.540867

(a) (b)

Figure 17: (a) Sparse model 7 digit specific classification confusion matrix. (b) Classification performance metrics.



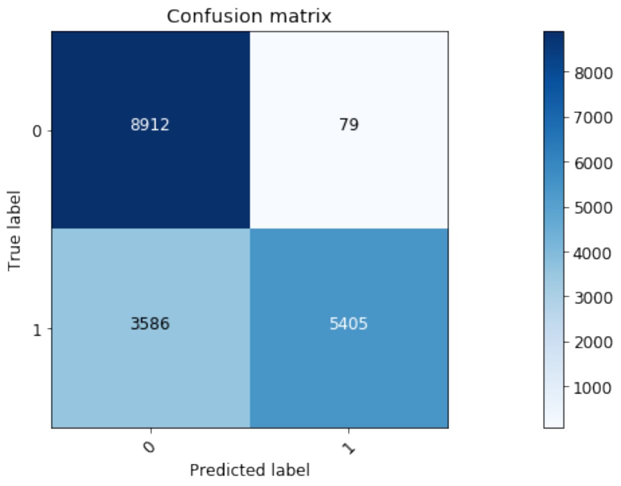
(a)



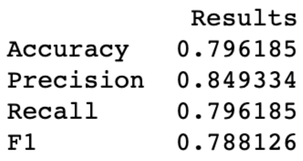
```
             Results
Accuracy    0.900288
Precision   0.900458
Recall      0.900288
F1          0.900277
```

(b)

Figure 18: (a) Sparse model 8 digit specific classification confusion matrix. (b) Classification performance metrics.



(a)



```
             Results
Accuracy    0.796185
Precision   0.849334
Recall      0.796185
F1          0.788126
```

(b)

Figure 19: (a) Sparse model 9 digit specific classification confusion matrix. (b) Classification performance metrics.

Appendix C Python Code

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


from mnist import MNIST
from os.path import expanduser
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.optimize import fmin
from sklearn import preprocessing
from matplotlib import rcParams
from sklearn import linear_model
from sklearn.multioutput import MultiOutputRegressor
import matplotlib.ticker as plticker
from matplotlib.patches import Patch
from tqdm.notebook import tqdm
from copy import deepcopy
import math
from matplotlib import gridspec
from PIL import Image
import pandas as pd
import random
import itertools
from scipy import interp
import matplotlib.cm as cm
from sklearn.metrics import *
from sklearn import preprocessing
from sklearn.multiclass import OneVsRestClassifier
from sklearn.utils import resample
from sklearn.model_selection import KFold


# In[8]:


'''boxPlotter takes in a 4D array X containing cross validated model coefficients for various model fit parameter
 values (e.g. alpha, or l1 ratio) and plots box plots for the cross validated data for eah coefficient value
 and colors these box plots by which column of x they come from. This serves to indcate not only which model
 coefficients have the highest values, but which of the multiple outputs they contribute to modeling.

    INPUTS:
    X: numpy.ndarray of data to analyze. Expected shape is (m,p,c,a) where m is the number of variables modelled,
        p is the number of outputs to regress on, c is the number of cross validation iterations,
        and a is the number of model fit parameters (e.g. alpha, l1 ratio) varied.
    categoreis: list of scalar values indicating model fit parameters to plot box plots for.
    top_perc: scalar value indicating the percent of largest coefficient values to plot. Default of 1%.
    medianbar: scalar value indicating the line width of the median bar in each box plot. Default of 3.
    fontsize: scalar value indicating the font size for each generated plot. Default of 12.
    figsize: list of two scalar values [w,h] indicating the width (w) and height (h) of each plot. As
            the number of top coefficients to plot increase the width of the plot should be adjusted
            accordingly. Default is [25, 5].
    class_title: string value that allows control over the title above each plot. Title appears as,
                Top %'+ str(top_perc)+ ' Model Coefficients by Digit Class, ' + class_title + str(categories[i])

    OUTPUTS:
    plots: sequence of box plots.

    NOTES:
    The color sequece used to discriminate columns of X for each coefficient can discriminate
    up to max of 10 categories.
'''
def boxPlotter(X, categories, top_perc=1.0, medianbar = 3, fontsize = 12, figsize = [25, 5], class_title='Var = '):
    m = X.shape[0]
    p = X.shape[1]
    multiple = len(categories) > 1
    for i in tqdm(range(0, len(categories))):
        if multiple:
            x = np.average(X[:,:,:,i], axis=2)
        else:
            x = np.average(X[:,:,:], axis=2)
        xflat = x.flatten()
        top = int(((m*p)*(top_perc/100)))
        ind = np.argpartition(xflat, -top, axis=0)[-top:]
        ind_sort = ind[np.argsort(xflat[ind])[::-1]]
        (pix, dig) = np.unravel_index(ind_sort, (m,p))
        if multiple:
            data = X[:,:,:,i][pix,dig,:]
        else:
            data = X[:,:,:][pix,dig,:]

        plt.rcParams['figure.figsize'] = figsize
        rcParams.update({'font.size': fontsize})
        medianprops = dict(linestyle='-.', linewidth=medianbar)
        fig, ax = plt.subplots()

        cols = []
        legend_elements = []

        for j in range(0, len(ind)):
            bp = ax.boxplot(data[j].T, positions = [j], medianprops=medianprops, patch_artist=True)
            color = 'C' + str(dig[j])

            for element in ['boxes', 'whiskers', 'fliers', 'means', 'medians', 'caps']:
                plt.setp(bp[element], color=color)
            for patch in bp['boxes']:
                patch.set(facecolor=color)

            if dig[j] not in cols:
                cols.append(dig[j])


        col_array = np.array(cols)
        cols_sorted = col_array[np.argsort(col_array)]
        for c in cols_sorted:
            color = 'C' + str(c)
            legend_elements.append(Patch(facecolor=color, edgecolor=color, label='coeff col ' + str(c)))

        ax.legend(handles=legend_elements, loc='best')
        ax.set_xticklabels(pix)
        ax.set_title('Top %'+ str(top_perc)+ ' Model Coefficients by Digit Class, ' + class_title + str(categories[i]) )



"""
This function prints and plots the confusion matrix.
Normalization can be applied by setting `normalize=True`.
Adapted from code at:
    http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
"""
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    bottom, top = plt.ylim()
    #plt.ylim(bottom + 0.5, top - 0.5)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()

'''Takes in a one hout encoded array where each column is a different class, and returns a 1D array with
the class value for each row represented by the index of the 1 hot encoding in that row.'''
def class_decoder(b):
    rows = b.shape[0]
    cols = b.shape[1]
    decoded = []
    for i in range(0, rows):
        maxValue = 0
        maxInd = 0
        for j in range(0, cols):
            if b[i, j] > maxValue:
                maxValue = b[i, j]
                maxInd = j
        decoded.append(maxInd)
    return decoded

'''Takes in an array of model coefficients finds the top terms up to the number of terms indicated (num_terms), keeps
those values, and zeros out the rest of the coefficient values. This allows for testing of model sparsity. If X is 3D
the third axis is interpreted as cross validated coefficient values. The coefficient to compare to find maximal
values is the average of coefficients across cross validation. The CV data should be along the 3rd data axis.'''
def zero_out(X, num_terms):
    dims = X.shape
    if len(dims) > 2:
        x = np.average(X, axis=2)
    else:
        x = X
    xflat = x.flatten()
    ind = np.argpartition(xflat,-(int(num_terms)))[-(int(num_terms)):]
    ind_sort = ind[np.argsort(xflat[ind])[::-1]]
    mask = np.zeros(len(xflat))
    mask[ind_sort] = 1
    xflat_z = xflat*mask
    x_zeroed = np.reshape(xflat_z, dims)
    return x_zeroed


# In[3]:


file = '/Documents/AMATH Masters/AMATH563/HW1/dataset/MNIST'
home = expanduser("~") + file
mnist = MNIST(home)
x_train_raw, y_train_raw = mnist.load_training() #60000 samples
x_test_raw, y_test_raw = mnist.load_testing()    #10000 samples

#convert lists to numpy arrays
x_train = np.asarray(x_train_raw) #60000 x 784 where 784 is flattened 28x28 pixel data
y_train = np.asarray(y_train_raw) #60000 x 1 where each entry is the digit class
x_test = np.asarray(x_test_raw) #60000 x 784 where 784 is flattened 28x28 pixel data
y_test = np.asarray(y_test_raw) #60000 x 1 where each entry is the digit class

#label binarize the class arrays, i.e. each column reprents a digit class
#with binary values representing whether or not the specific image is of that digit class
lb = preprocessing.LabelBinarizer()
lb.fit(y_train)
y_train = lb.transform(y_train)# 60000 x 10 where ach column is a digit class (0 through 9)

lb = preprocessing.LabelBinarizer()
lb.fit(y_test)
y_test = lb.transform(y_test)# 10000 x 10 where each column is a digit class (0 through 9)

n = x_train.shape[0] #number of images in data set (training set has 60000 flattened images)
m = x_train.shape[1] #number of pixels (28x28 yields 784 pixels total)
p = y_train.shape[1] #number of classes (10 classes, 0 through 9)
A = x_train
b = y_train


# In[4]:


lam = 0.1
cv_folds = 5
kf = KFold(n_splits = cv_folds, random_state = 17, shuffle = True)

E1 = np.zeros(cv_folds)
E2 = np.zeros(cv_folds)
E3 = np.zeros(cv_folds)
E4 = np.zeros(cv_folds)
E5 = np.zeros(cv_folds)
E6 = np.zeros(cv_folds)

X1 = np.zeros((m,p,cv_folds))
X2 = np.zeros((m,p,cv_folds))
X3 = np.zeros((m,p,cv_folds))
X4 = np.zeros((m,p,cv_folds))
X5 = np.zeros((m,p,cv_folds))
X6 = np.zeros((m,p,cv_folds))
jj = 0
```

```
for train_index, test_index in kf.split(A):
    print("Progress {:2.1%}".format(jj / cv_folds), end="\r")#print progress of paint can position acquisition
    #print("Progress {:2.1%}".format(jj / cv_folds), end="\r")#print progress
    x1 = np.linalg.pinv(A[train_index,:]) @ b[train_index,:]
    b1 = A[train_index,:] @ x1
    E1[jj] = np.linalg.norm(b[train_index,:] - b1,ord=2)/np.linalg.norm(b[train_index,:],ord=2)

    x2 = np.linalg.lstsq(A[train_index,:],b[train_index,:],rcond=None)[0]
    b2 = A[train_index,:] @ x2
    E2[jj] = np.linalg.norm(b[train_index,:]-b2,ord=2)/np.linalg.norm(b[train_index,:],ord=2)

    regr3 = linear_model.ElasticNet(alpha=1.0, copy_X=True, l1_ratio=lam, max_iter=10**5,random_state=0)
    regr3.fit(A[train_index,:], b[train_index,:])
    x3 = np.transpose(regr3.coef_)
    b3 = A[train_index,:] @ x3
    E3[jj] = np.linalg.norm(b[train_index,:]-b3,ord=2)/np.linalg.norm(b[train_index,:],ord=2)

    regr4 = linear_model.ElasticNet(alpha=0.8, copy_X=True, l1_ratio=lam, max_iter=10**5,random_state=0)
    regr4.fit(A[train_index,:], b[train_index,:])
    x4 = np.transpose(regr4.coef_)
    b4 = A[train_index,:] @ x4
    E4[jj] = np.linalg.norm(b[train_index,:]-b4,ord=2)/np.linalg.norm(b[train_index,:],ord=2)

    regr5 = MultiOutputRegressor(linear_model.HuberRegressor(), n_jobs=-1)
    huber = regr5.fit(A[train_index,:], b[train_index,:]) # matlab's robustfit() does not have an exact sklearn analogue

    x5 = np.empty([m, p])
    for i in range(0, len(huber.estimators_)):
        x5[:, i] = huber.estimators_[i].coef_

    b5 = A[train_index,:] @ x5
    E5[jj] = np.linalg.norm(b[train_index,:]-b5,ord=2)/np.linalg.norm(b[train_index,:],ord=2)

    ridge = linear_model.Ridge(alpha=1.0).fit(A[train_index,:],b[train_index,:])
    x6 = np.transpose(ridge.coef_)
    b6 = A[train_index,:] @ x6
    E6[jj] = np.linalg.norm(b[train_index,:] - b6,ord=2)/np.linalg.norm(b[train_index,:],ord=2)

    X1[:,:,jj] = x1
    X2[:,:,jj] = x2
    X3[:,:,jj] = x3
    X4[:,:,jj] = x4
    X5[:,:,jj] = x5
    X6[:,:,jj] = x6

    jj = jj + 1


Err = np.column_stack((E1,E2,E3,E4,E5,E6))

reg_styles = ['pinv', 'lstsq', 'elastic (alpha=1)', 'elastic (alpha=0.8)', 'huber', 'ridge']#types of Ax=b solvers used
Xdict = [X1, X2, X3, X4, X5, X6]#list of model coefficients for plotting

plt.rcParams['figure.figsize'] = [15, 21]
rcParams.update({'font.size': 12})
fig,axs = plt.subplots(len(reg_styles),1, sharex=True)
axs = axs.reshape(-1)

for j in range(0, len(reg_styles)):
    x = np.average(Xdict[j], axis=2)#average across cross validated coefficient values
    x_pcolor = axs[j].pcolor(x.T,cmap='Greys', vmin=-0.06, vmax=0.16)
    fig.colorbar(x_pcolor, ax=axs[j])
    axs[j].set_ylabel('Digit Class')
    axs[j].set_title(reg_styles[j])
    plt.setp(axs[j].get_xticklabels(), visible=False)

plt.rcParams['figure.figsize'] = [8, 8]

plt.figure()
plt.boxplot(Err)
plt.xticks([1, 2, 3, 4, 5, 6], ['pinv', 'np.linalg.solve', 'Elastic 1.0', 'Elastic 0.8', 'Huber', 'Ridge'], rotation=40)
plt.show()


# In[5]:


top_perc = 0.5 #top percent of coefficients to extract
Xdict4D = np.transpose(np.array(Xdict), (1, 2, 3, 0))#change Xdict from list of 3D arrays to 4D array

boxPlotter(Xdict4D, reg_styles, top_perc=0.5, medianbar=3, fontsize=12, figsize=[20, 5], class_title='Regression Style: ')
```

```
# In[7]:


#Work with elastic net to test out different l1_ratios to promote sparsity
cv_folds = 5
kf = KFold(n_splits = cv_folds, random_state = 17, shuffle = True)
l1_ratios = [1.0, 0.75, 0.5, 0.25]
Esparse = np.zeros((cv_folds, len(l1_ratios)))
Xsparse = np.zeros((m,p,cv_folds,len(l1_ratios)))

i = 0
for train_index, test_index in kf.split(A):
    print("Progress {:2.1%}".format(i / cv_folds), end="\r")#print progress of paint can position acquisition
    for j in range(0, len(l1_ratios)):
        regr_sparse = linear_model.ElasticNet(alpha=1.0, copy_X=True, l1_ratio=l1_ratios[j], max_iter=10**5,random_state=0)
        regr_sparse.fit(A[train_index,:], b[train_index,:])
        xsparse = np.transpose(regr_sparse.coef_)
        bsparse = A[train_index,:] @ xsparse
        Esparse[i, j] = np.linalg.norm(b[train_index,:]-bsparse,ord=2)/np.linalg.norm(b[train_index,:],ord=2)
        Xsparse[:,:,i,j] = xsparse
    i = i + 1#iterate cross validation counter

boxPlotter(Xsparse, l1_ratios, top_perc=0.5, medianbar=3, fontsize=12, figsize=[20, 5], class_title='l1 ratio: ')


# In[11]:


#Work with elastic net to test out different alpha values to promote sparsity
cv_folds = 5
kf = KFold(n_splits = cv_folds, random_state = 17, shuffle = True)
Elasso = np.zeros((cv_folds, len(l1_ratios)))
Xlasso = np.zeros((m,p,cv_folds,len(l1_ratios)))
alphas = [0.25, 0.5, 0.75, 1.0]

i = 0
for train_index, test_index in kf.split(A):
    print("Progress {:2.1%}".format(i / cv_folds), end="\r")#print progress of paint can position acquisition
    for j in range(0, len(alphas)):
        lasso = linear_model.ElasticNet(alpha=alphas[j], copy_X=True, l1_ratio=1.0, max_iter=10**5,random_state=0)
        lasso.fit(A[train_index,:], b[train_index,:])
        xlasso = np.transpose(lasso.coef_)
        blasso = A[train_index,:] @ xlasso
        Elasso[i, j] = np.linalg.norm(b[train_index,:]-blasso,ord=2)/np.linalg.norm(b[train_index,:],ord=2)
        Xlasso[:,:,i,j] = xlasso
    i = i + 1

boxPlotter(Xlasso, alphas, top_perc=0.5, medianbar=3, fontsize=12, figsize=[20, 5], class_title='Alpha: ')


# In[7]:


#Apply your most important pixels to the test data set to see how accurate you are with as few pixels as possible.
#Best l1 ratio appears to be 1.0 and best alpha value appears to be 0.25. Try a model employing both values.
#Work with elastic net to test out different alpha values to promote sparsity
cv_folds = 5
kf = KFold(n_splits = cv_folds, random_state = 17, shuffle = True)
l1_ratio = 1.0
Elasso_best = np.zeros((cv_folds))
Xlasso_best = np.zeros((m,p,cv_folds))
alpha = 0.25

i = 0
for train_index, test_index in kf.split(A):
    print("Progress {:2.1%}".format(i / cv_folds), end="\r")#print progress of paint can position acquisition
    lasso_best = linear_model.ElasticNet(alpha=alpha, copy_X=True, l1_ratio=l1_ratio, max_iter=10**5,random_state=0)
    lasso_best.fit(A[train_index,:], b[train_index,:])
    xlasso_best = np.transpose(lasso_best.coef_)
    blasso_best = A[train_index,:] @ xlasso_best
    Elasso_best[i] = np.linalg.norm(b[train_index,:]-blasso_best,ord=2)/np.linalg.norm(b[train_index,:],ord=2)
    Xlasso_best[:,:,i] = xlasso_best
    i = i + 1

boxPlotter(Xlasso_best, [alpha], top_perc=0.5, medianbar=3, fontsize=12, figsize=[20, 5], class_title='Alpha: ')

Atest = x_test
btest = y_test
classes = np.unique(y_test_raw)
```

```
preds = lasso_best.predict(Atest)#test the model
preds_1D = np. array(class_decoder(preds))
btest_1D = np. array(class_decoder(btest))

CM = confusion_matrix(btest_1D, preds_1D)# Confusion Matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix

aScore = accuracy_score(btest_1D, preds_1D)#accuracy score
P = precision_score(btest_1D, preds_1D, average='weighted')#precision score
R = recall_score(btest_1D, preds_1D, average='weighted')#recall score
F1 = f1_score(btest_1D, preds_1D, average='weighted')#F1 score
a = {'Results': [aScore, P, R, F1]}#series of evaluation results
aFrame_k = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_k)


# In[151]:


#perform analysis for degree of sparsity for best lasso model
num_terms = [1, 5, 10, 25, 50, 150, 300, 600]
xlasso_bz_plot = np.empty((m, p, cv_folds, len(num_terms)))
xlasso_bz = np.empty((m, p,len(num_terms)))
for i in range(0, len(num_terms)):

    xlasso_bz[:,:,i] = zero_out(np.average(Xlasso_best[:,:,:], axis=2), num_terms[i])#zero out coefficients for average cv lasso model
    mask = np.ones(xlasso_bz[:,:,i].shape)*xlasso_bz[:,:,i]
    mask = np.repeat(mask[:, :, np.newaxis], 5, axis=2)

    xlasso_bz_plot[:,:,:,i] = mask*Xlasso_best[:,:,:]

boxPlotter(xlasso_bz_plot, num_terms, top_perc=0.5, medianbar=3, fontsize=12, figsize=[20, 5], class_title='Number of Terms: ')

for i in range(0, len(num_terms)):

    Atest = x_test
    btest = y_test
    classes = np.unique(y_test_raw)
    preds =  Atest @ xlasso_bz[:,:,i]#test the model
    preds_1D = np. array(class_decoder(preds))
    btest_1D = np. array(class_decoder(btest))

    CM = confusion_matrix(btest_1D, preds_1D)# Confusion Matrix
    plot_confusion_matrix(CM, classes, title='Confusion matrix: ' + str(num_terms[i]) + ' Model Terms')#plot of the multiclass confusion matrix


    aScore = accuracy_score(btest_1D, preds_1D)#accuracy score
    P = precision_score(btest_1D, preds_1D, average='weighted')#precision score
    R = recall_score(btest_1D, preds_1D, average='weighted')#recall score
    F1 = f1_score(btest_1D, preds_1D, average='weighted')#F1 score
    a = {'Results': [aScore, P, R, F1]}#series of evaluation results
    aFrame_k = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
    print(aFrame_k)


# In[37]:


#Redo analysis with each digit individually
cv_folds = 5
kf = KFold(n_splits = cv_folds, random_state = 17, shuffle = True)
classes = np.unique(y_test_raw)
l1_ratio = 1.0
Elasso_dig = np.zeros((len(classes), cv_folds))
Xlasso_dig = np.zeros((m,1,len(classes), cv_folds))
alpha = 0.25
top_perc=5
medianbar=3
fontsize=12
figsize=[20, 5]
class_title='Alpha: '

for i in tqdm(range(0, len(classes))):
    #Need to up sample minority digit class to resolve class imbalance problem
    btrain = b[:, i]
    dig_ind = np.where(btrain == 1)[0]
    other_ind = np.where(btrain == 0)[0]
    upsampled_b = resample(btrain[dig_ind], replace=True,n_samples=len(other_ind),random_state=123)
    upsampled_A = resample(A[dig_ind], replace=True,n_samples=len(other_ind),random_state=123)
    btrain = np.append(btrain[other_ind], upsampled_b, axis=0)
    Atrain = np.append(A[other_ind,:], upsampled_A, axis=0)
    j = 0
    for train_index, test_index in kf.split(Atrain):
    #perform cross validation per digit class
        lasso_dig = linear_model.ElasticNet(alpha=alpha, copy_X=True, l1_ratio=l1_ratio, max_iter=10**5,random_state=0)
        lasso_dig.fit(Atrain[train_index,:], btrain[train_index])
        xlasso_dig = np.transpose(lasso_dig.coef_)
        blasso_dig = Atrain[train_index,:] @ xlasso_dig
        Elasso_dig[i, j] = np.linalg.norm(btrain[train_index]-blasso_dig,ord=2)/np.linalg.norm(btrain[train_index],ord=2)
        Xlasso_dig[:,:,i,j] = np.expand_dims(xlasso_dig, axis=1)
        j = j + 1

for i in tqdm(range(0, len(classes))):
    m = Xlasso_dig.shape[0]
    p = Xlasso_dig.shape[1]
    xav = np.average(Xlasso_dig[:,:,i,:], axis=2)
    xflat = xav.flatten()
    '''
    top = int(((m*p)*(top_perc/100)))
    top_inds = np.argpartition(xflat, -top, axis=0)[-top:]
    top_vals = xflat[top_inds]
    data = top_vals[np.argsort(top_vals)[::-1]]
    '''
    ind = np.argpartition(xflat, -top, axis=0)[-top:]
    ind_sort = ind[np.argsort(xflat[ind])[::-1]]
    (pix, dig) = np.unravel_index(ind_sort, (m,p))

    data = Xlasso_dig[:,:,i,:][pix,dig,:]
    plt.rcParams['figure.figsize'] = figsize
    rcParams.update({'font.size': fontsize})
    medianprops = dict(linestyle='-.', linewidth=medianbar)
    fig, ax = plt.subplots()

    cols = []
    legend_elements = []

    for j in range(0, len(top_vals)):
        bp = ax.boxplot(data[j], positions = [j], medianprops=medianprops, patch_artist=True)
        color = 'C' + str(i)

        for element in ['boxes', 'whiskers', 'fliers', 'means', 'medians', 'caps']:
            plt.setp(bp[element], color=color)
        for patch in bp['boxes']:
            patch.set(facecolor=color)

    ax.set_xticklabels(top_inds[np.argsort(top_vals)[::-1]])
    ax.set_title('Top %'+ str(top_perc)+ ' Model Coefficients for Digit Class, ' + str(classes[i]) )


    #upsample minority class in test data
    Atest = x_test
    btest = y_test[:, i]
    dig_ind = np.where(btest == 1)[0]
    other_ind = np.where(btest == 0)[0]
    upsampled_b = resample(btest[dig_ind], replace=True,n_samples=len(other_ind),random_state=123)
    upsampled_A = resample(Atest[dig_ind], replace=True,n_samples=len(other_ind),random_state=123)
    btest_itr = np.append(btest[other_ind], upsampled_b, axis=0)
    Atest_itr = np.append(Atest[other_ind,:], upsampled_A, axis=0)

    preds_itr =  Atest_itr @ xav#test the model
    for k in range(0, len(preds_itr)):
        if preds_itr[k] >=0.55:
            preds_itr[k] = 1
        else:
            preds_itr[k] = 0

    preds_itr = preds_itr.astype(int)
    CM = confusion_matrix(btest_itr, preds_itr)# Confusion Matrix
    plot_confusion_matrix(CM, [0, 1])#plot of the multiclass confusion matrix


    aScore = accuracy_score(btest_itr, preds_itr)#accuracy score
    P = precision_score(btest_itr, preds_itr, average='weighted')#precision score
    R = recall_score(btest_itr, preds_itr, average='weighted')#recall score
    F1 = f1_score(btest_itr, preds_itr, average='weighted')#F1 score
    a = {'Results': [aScore, P, R, F1]}#series of evaluation results
    aFrame_k = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
    print(aFrame_k)
```