1-Norm Solvers and Solution Sparsity

Maple Tan
AMATH563 Homework 1

April 19, 2019

1 Abstract

Exploration of hand written numbers from the MNIST dataset. Specifically exploring sparse mappings that map the images to their respective labels, and how to achieve those sparse mappings.

2 Introduction and Overview

Often times data science encounters the problem $Ax = b$, where A can be a matrix of features, b is the target feature and x is the solution that maps the two. When working with real life data, this almost always creates overdetermined or underdetermined systems. In other words, we will have systems that have more than one solution (overdetermined) or systems that do not have any (underdetermined). In the case of overdetermined systems, the question becomes what would be the 'best' solution that solves our system. The goal of this assignment is to understand and explore the different ways one can solve overdetermined systems and determine which ones would be most appropriate to use.

The dataset in question is called the MNIST dataset, and it contains a training and testing set of 28 x 28 pixel images of handwritten numbers reshaped into column vectors and their respective labels. This assignment aims to find a mapping from the image space to the label space and also determine/rank which pixels are most important in correctly labeling the images.

3 Theoretical Background

In a typical optimization problem, suppose we have a system defined as

$$Ax = b \tag{1}$$

Where A is our data matrix, b is our labels matrix and x is the mapping that maps from one space to the other that is solved for. When solving for x, often the 2-norm is used as a metric to determine how good a solution is. In particular, we want to minimize the least

squared error between the Ax and b. In the case of overdetermined systems, we would want to minimize the below equation:

$$\arg\min_x \left[||Ax - b|| + \lambda g(x) \right] \tag{2}$$

Where $\lambda g(x)$ is a regularization term. The regularization term can be used to promote certain aspects we want to see in a solution, such as sparsity. In particular, promoting sparsity can help determine which features are most important in classification, where the nonzero entries of the solution indicate importance.

The main issue that comes with the 2-norm however is that it is extremely sensitive to outliers by construction. In fact, a single outlier could completely throw off the solution to an optimization problem. A solution to this problem is to use the 1-norm instead, which is much more robust against outliers and also has the property of promoting sparsity within the solution. While the 2-norm minimizes the solution (i.e. makes the answer as small as possible but not necessarily 0), the 1-norm promotes a solution to have as many zero entries as possible.

4 Algorithm Implementation and Development

Starting off, we must import all the necessary libraries in Python, this includes `numpy`, `matplotlib`, `sklearn` and `mnist`. We must then load the necessary data from the compressed files downloaded from the mnist site. The `mnist` library provides an easy way to decompress and load the data. It has methods such as `load_training` and `load_testing` that returns a list of 1-D vectors containing the images and a vector of numbers indicating which digit the image corresponds to. After loading the data, we now have to we have to construct our A and b matrices as per the problem description. Starting with A, we have:

$$A = [x_1 \;\; x_2 \;\; x_3 \;\; \cdots \;\; x_n] \tag{3}$$

Where x_j are column vectors of jth original reshaped 28 x 28 images and n is the number of images in the data set. Then next for b we have:

$$B = [y_1 \;\; y_2 \;\; y_3 \;\; \cdots \;\; y_n] \tag{4}$$

Where y_j are the labels that indicate which digit the image x_j corresponds to. The columns of B are constructed as a 10 x 1 vector, where we set the element that corresponds to the digit to 1 and the rest to 0. In the case of the label being 0, we set 1 to the 10th element, as shown below:

$$\text{"1"} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \text{"2"} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \text{"3"} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \cdots \text{"9"} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \text{"0"} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

After creating these matrices, we also need to take the transpose of both A and B in order to solve for x. In the case of our training set, since there are 60,000 images, this results in the following dimensions for our system.

$$\begin{aligned} A^T \ x &= b^T \\ [60000 \times 784][784 \times 10] &= [60000 \times 10] \end{aligned}$$

Next, we look at different ways to solve this system. For 2-norm solvers, we use methods such as `pinv` and `ridge` from the `numpy` and `sklearn` library. Then for 1-norm solvers, we use the `lasso` method from the `sklearn` library.

Now we use the sparse x (`lasso` method) we solved for to determine the most important pixels in our image. Intuitively, the 0s in our solution imply less importance while higher magnitude numbers imply more importance. As such, we take the sum of the absolute value of x across its columns, giving us 1×784 vector. This vector contains weights for each of the 784 pixels. Next, we use the `argsort` method from the `numpy` library, which returns a vector that is used to sort a vector from least to great in term in magnitude. We then use that to create a boolean matrix where the top n important pixels are set to 1 and then the rest to 0. With this, we can then apply this pixel importance matrix to each column of our testing matrix, which would set the 'non-important' pixels to 0. We then repeat all of this but then separate the training set by digits and multiplying it by the corresponding column of x, which can also be interpreted as a vector of weights.

To help determine how well the pixels can predict a label, we will apply KNearestNeighbors and DecisionTree to the modified training and testing data to see how well the two algorithms can predict a digit using only those pixels.

5 Computational Results

After solving the systems, one can interpret x, which is a 784×10, as a bunch of weights. The larger the magnitude of the weight at position (a, b), the more important that pixel in row a is in associating the image with that specific digit in column b. This means that when we multiply Ax, we see that a row of A, which is a reshaped image, is multiplied by these pixel weights for each digit. Then if the image contains the right pixels that is associated with each digit, the resulting vector in b would correspond with the correct label. Then calculating for x can yield the following results depending on the method:

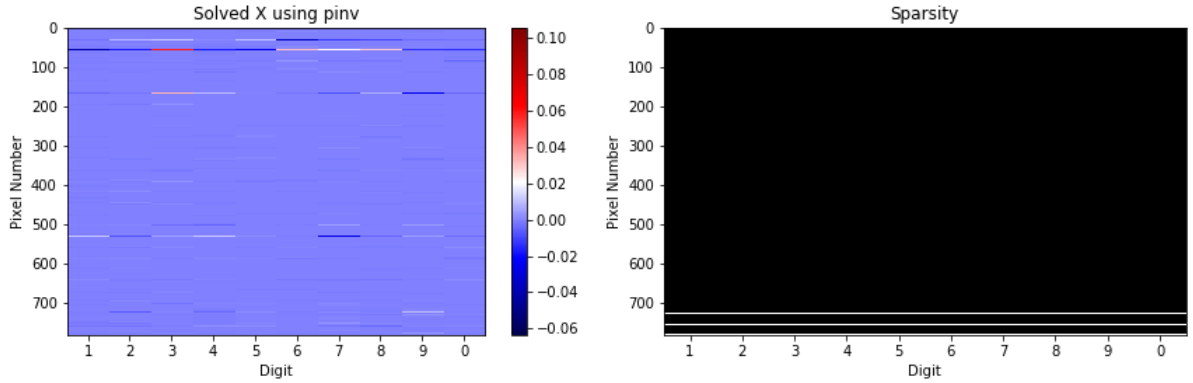

Figure 1: In the above graph on the right, white shows that an element of our solution is 0 and black shows that it is non-0. The pinv method uses a 2-norm/least squared method to solve our system, we see that the a lot of the values are fairly small and close to 0 but very few actually equaling 0.

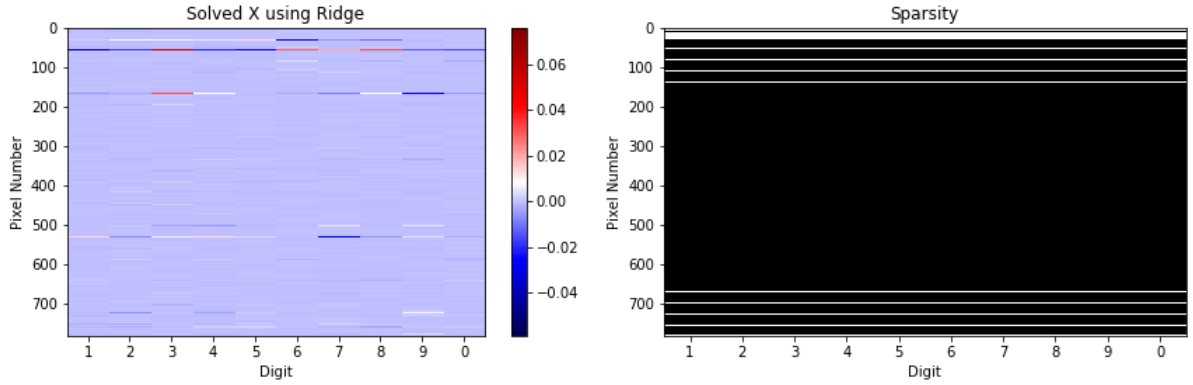

Figure 2: The ridge method also uses a 2-norm solver. Though, different from pinv in that it has some more sparsity, we see that a lot of the elements are still nonzero.

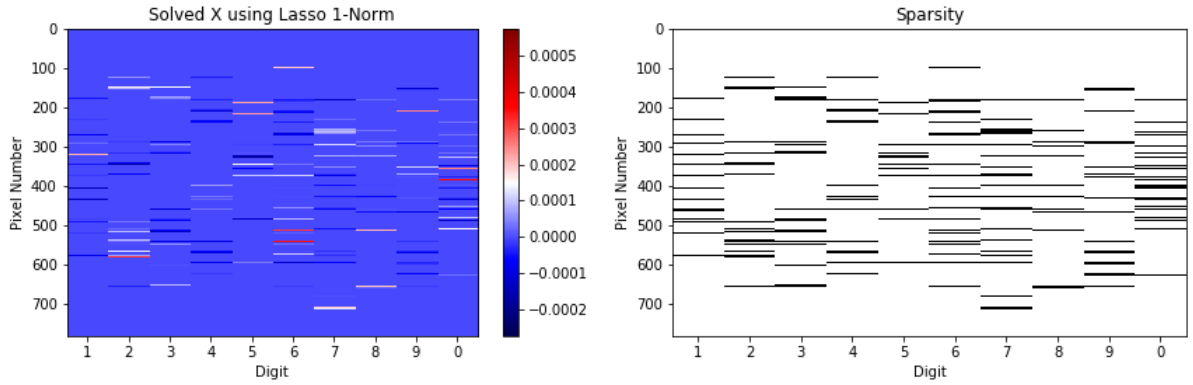

Figure 3: In this case, the lasso method uses the 1-norm. As stated before, since sparsity is a property of using a 1-norm solver, there are many more elements that equal 0.

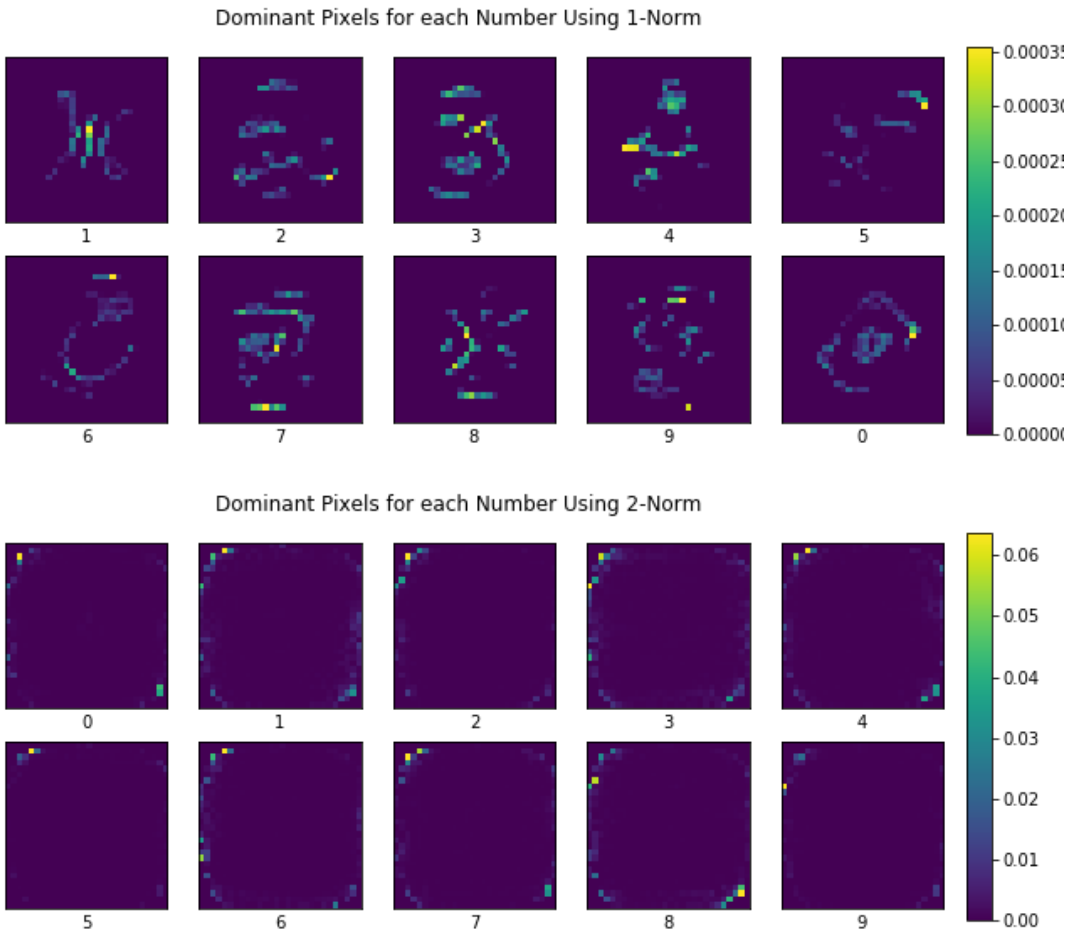

Figure 4: Here we plot each column of our solution x where each column corresponds with a digit. We see that the 1-norm solver, due to its sparsity, gives us a better general shape of the different digits in the solution while the 2-norm solver does not.

When looking at how well classification works with only the important pixels, we see similar yielded accuracy on the modified test data relative to the original accuracy in both the KNearestNeighbors and NearestTree methods. Applying the summed up weights of the pixels across each digit is on par in terms of accuracy with the accuracy of the original while applying the corresponding x column does yield a relatively high accuracy with KNearestNeighbors but is noticeably lower than the other two.

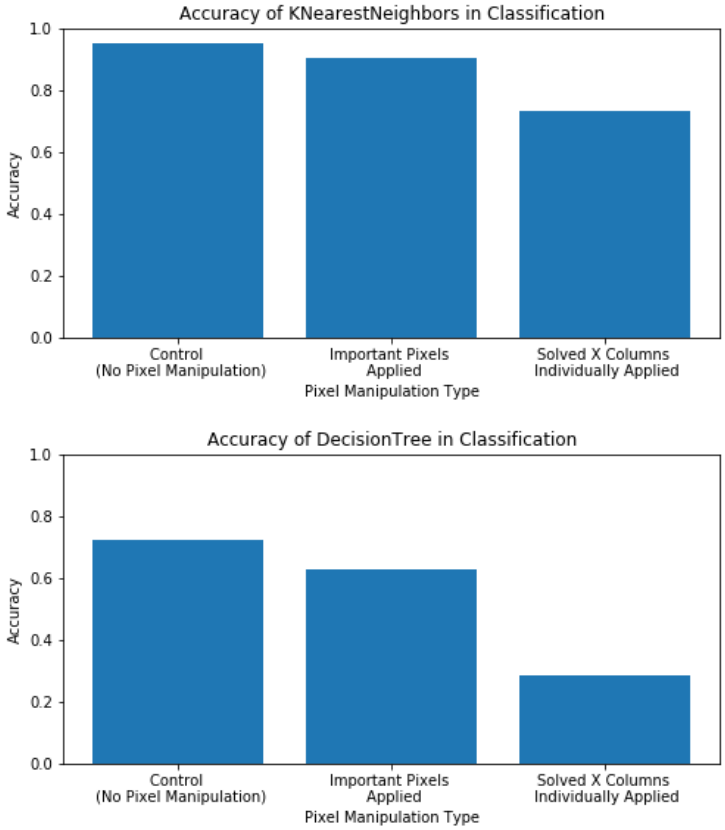

Figure 5: The accuracy of KNearestNeighbors and DecisionTree classifiers using the modified dataset and original dataset (control)

6 Summary and Conclusions

Promoting sparsity is especially important in overdetermined systems. Due to the possibility of multiple answers, it becomes extremely important to know what kind of answer we want when solving the system. In the case of this problem, we see that promoting sparsity helps determine how important a pixel is in categorizing the image to a digit. In this way, we see that certain pixels have more of an impact on certain categorizing than others.

A Python Code

```
#%%
import numpy as np
import numpy.linalg as la
from mnist import MNIST
import matplotlib.pyplot as plt

def return_labels_mat(lab_vals, num):
    lab = np.zeros([10,num])
    k = 0
    for y in lab_vals:
        lab[y-1][k] = 1
        k = k+1
    return lab

def return_image_mat(im_list, num):
    i = im_list[0]
    im = np.zeros([np.size(i), num])
    k = 0
    for i in im_list:
        im[:,k] = i
        k = k+1
    return im

def create_imp_pixels_mat_binary(matx, nn):
    X_sum = np.sum(abs(matx), axis=1)
    Max_inx = np.argsort(X_sum.flatten())
    Imp_pix = np.squeeze(np.asarray(Max_inx < nn)).astype(int)
    return Imp_pix

def get_label(label):
    lab = label.index(1) + 1
    if lab == 10:
        lab = 0
    return lab


#%%
mndata = MNIST('Data')
mndata.gz = True

exa = [10000, 60000] # Number of smaples in testing and training data respectively
images_train, labels_train = mndata.load_training()
```

```
images_test, labels_test = mndata.load_testing()
B_train = np.transpose(return_labels_mat(labels_train, exa[1]))
A_train = np.transpose(return_image_mat(images_train, exa[1]))
B_test = np.transpose(return_labels_mat(labels_test, exa[0]))
A_test = np.transpose(return_image_mat(images_test, exa[0]))


# %% Create Initial classifier
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors = 10)
neigh.fit(A_train,B_train)

#%%
score = neigh.score(A_test, B_test)
print(score)

# score = 0.9558
# %%
# Use numpy.linalg.pinv - 2 norm to solve system

X2n_train = la.pinv(A_train)@B_train
X2n_test = la.pinv(A_test)@B_test
X2n_sp = np.abs(X2n_train) == 0

#%% Plot pixel sparsity
fig, ax = plt.subplots(1,2, figsize=(12,4))
plt0 = ax[0].imshow(X2n_train, aspect='auto', cmap='seismic')
ax[0].set_xticks(np.arange(0,10))
ax[0].set_xticklabels([1,2,3,4,5,6,7,8,9,0])
ax[0].set_ylabel('Pixel Number')
ax[0].set_xlabel('Digit')
ax[0].set_title('Solved X using pinv')
fig.colorbar(plt0, ax=ax[0])
plt1 = ax[1].imshow(X2n_sp, aspect='auto', cmap='gray')
ax[1].set_xticks(np.arange(0,10))
ax[1].set_xticklabels([1,2,3,4,5,6,7,8,9,0])
ax[1].set_ylabel('Pixel Number')
ax[1].set_xlabel('Digit')
ax[1].set_title('Sparsity')
plt.tight_layout()
plt.savefig('pinv.png')
plt.show()

#%%
```

```
# Use numpy.linalg.lstsq - minimizes least square - 2 norm to solve system
from sklearn.linear_model import Ridge
clf = Ridge(alpha=1.0)
clf.fit(A_train, B_train)
Xridge = np.transpose(clf.coef_)
Xrid_sp = np.abs(Xridge) == 0

# %%
fig, ax = plt.subplots(1,2, figsize=(12,4))
plt0 = ax[0].imshow(Xridge, aspect='auto', cmap='seismic')
ax[0].set_xticks(np.arange(0,10))
ax[0].set_xticklabels([1,2,3,4,5,6,7,8,9,0])
ax[0].set_ylabel('Pixel Number')
ax[0].set_xlabel('Digit')
ax[0].set_title('Solved X using Ridge')
fig.colorbar(plt0, ax=ax[0])
plt1 = ax[1].imshow(Xrid_sp, aspect='auto', cmap='gray')
ax[1].set_xticks(np.arange(0,10))
ax[1].set_xticklabels([1,2,3,4,5,6,7,8,9,0])
ax[1].set_ylabel('Pixel Number')
ax[1].set_xlabel('Digit')
ax[1].set_title('Sparsity')
plt.tight_layout()
plt.savefig('ridge.png')
plt.show()

# %% Plot the dominant pixels found through 2-norm, shows general shape of
#   digits as per decided by 2-norm
fig, ax = plt.subplots(2,5, figsize=(10,4))
i = 0
j = 0
for num in np.arange(0,10):
    im = np.reshape(abs(X2n_train[:,num]), [28,28])
    img = ax[i][j].imshow(im, aspect='auto')
    ax[i][j].set_xticks([])
    ax[i][j].set_yticks([])
    ax[i][j].set_xlabel(num)
    j = j + 1
    if j == 5:
        i = i + 1
        j = 0
cb_ax = fig.add_axes([0.92, 0.1, 0.02, 0.8])
cbar = fig.colorbar(img, cax=cb_ax)
plt.suptitle('Dominant Pixels for each Number Using 2-Norm')
```

```
plt.savefig('x_2norm.png')
plt.show()


#%% Solve system using 1-norm
from sklearn import linear_model as lm
clf = lm.Lasso(alpha=1.5)
clf.fit(A_train,B_train)

# 1-norm
X1n_train = clf.sparse_coef_
X1n_train = np.transpose(X1n_train.todense())
X1n_sp = np.abs(X1n_train) == 0


# %%
# Plot
fig, ax = plt.subplots(1,2, figsize=(12,4))
plt0 = ax[0].imshow(X1n_train, aspect='auto', cmap='seismic')
ax[0].set_xticks(np.arange(0,10))
ax[0].set_xticklabels([1,2,3,4,5,6,7,8,9,0])
ax[0].set_ylabel('Pixel Number')
ax[0].set_xlabel('Digit')
ax[0].set_title('Solved X using Lasso 1-Norm')
fig.colorbar(plt0, ax=ax[0])
plt1 = ax[1].imshow(X1n_sp, aspect='auto', cmap='gray')
ax[1].set_xticks(np.arange(0,10))
ax[1].set_xticklabels([1,2,3,4,5,6,7,8,9,0])
ax[1].set_ylabel('Pixel Number')
ax[1].set_xlabel('Digit')
ax[1].set_title('Sparsity')
plt.tight_layout()
plt.savefig('lasso.png')
plt.show()


# %% # %% Plot the dominant pixels found through 1-norm, shows general shape of
#   digits as per decided by 1-norm
fig, ax = plt.subplots(2,5, figsize=(10,4))
i = 0
j = 0
pix_nums = [1,2,3,4,5,6,7,8,9,0]
for num in np.arange(0,10):
    im = np.reshape(abs(X1n_train[:,num]), [28,28])
```

```
    img = ax[i][j].imshow(im, aspect='auto')
    ax[i][j].set_xticks([])
    ax[i][j].set_yticks([])
    ax[i][j].set_xlabel(pix_nums[num])
    j = j + 1
    if j == 5:
        i = i + 1
        j = 0
cb_ax = fig.add_axes([0.92, 0.1, 0.02, 0.8])
cbar = fig.colorbar(img, cax=cb_ax)
plt.suptitle('Dominant Pixels for each Number Using 1-Norm')
plt.savefig('x_1norm.png')
plt.show()
# %% Apply most 'important' pixels to original image
# Recreate Data based on 'imporant pixe;s'
pixs = np.arange(50,750, 50)
x_imp = create_imp_pixels_mat_binary(X1n_train, 250)
A_train_imp = x_imp*A_train
A_test_imp = x_imp*A_test

# %%
neigh = KNeighborsClassifier(n_neighbors = 10)
neigh.fit(A_train_imp,B_train)
score_imp_pix_only = neigh.score(A_test_imp, B_test)
print(score_imp_pix_only)

# score_imp_pix_only = 0.350 top 250 pixels
# score_imp_pix_only = 0.5945 when increased to top 400 pixels

#%%


# %%
A_test_ind_pix = np.zeros([exa[0], 784])
for n in np.arange(0,exa[0]):
    label = list(B_test[n][:]).index(1)
    A_test_ind_pix[n][:] = np.multiply(np.transpose(X1n_train[:,label]),A_test[n,:])

A_train_ind_pix = np.zeros([exa[1], 784])
for n in np.arange(0,exa[0]):
    label = list(B_test[n][:]).index(1)
    A_train_ind_pix[n][:] = np.multiply(np.transpose(X1n_train[:,label]),A_train[n,:])

# %%
```

```
neigh = KNeighborsClassifier(n_neighbors = 10)
neigh.fit(A_train_ind_pix,B_train)
score_imp_pix_ind = neigh.score(A_test_ind_pix, B_test)
print(score_imp_pix_ind)

# %%
from sklearn import tree
clf = tree.DecisionTreeRegressor()
clf = clf.fit(A_train, B_train)
scoret = clf.score(A_test, B_test)

clf = tree.DecisionTreeRegressor()
clf = clf.fit(A_train_imp, B_train)
scoreta = clf.score(A_test_imp, B_test)

clf = tree.DecisionTreeRegressor()
clf = clf.fit(A_train_ind_pix, B_train)
scoretb = clf.score(A_test_ind_pix, B_test)


# %%
plt.figure(figsize=(8,4))
x_labels = ['Control \n (No Pixel Manipulation)', ...
 'Important Pixels \n Applied', ...
  'Solved X Columns \n Individually Applied']
plt.bar(x_labels, [score, score_imp_pix_only, score_imp_pix_ind])
plt.ylabel('Accuracy')
plt.xlabel('Pixel Manipulation Type')
plt.title('Accuracy of KNearestNeighbors in Classification')
plt.gcf().subplots_adjust(bottom=0.15)
plt.savefig('c.png')
plt.show()

# %%
plt.figure(figsize=(8,4))
x_labels = ['Control \n (No Pixel Manipulation)', ...
'Important Pixels \n Applied', ...
'Solved X Columns \n Individually Applied']
plt.bar(x_labels, [scoret, scoreta, scoretb])
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.xlabel('Pixel Manipulation Type')
plt.title('Accuracy of DecisionTree in Classification')
plt.gcf().subplots_adjust(bottom=0.15)
```

```
plt.savefig('d.png')
plt.show()
```