# Brunton and Kutz Problem 4.4: MNIST Classification with Multilinear Regression

Source Filename: /main.py

Rico A. R. Picone

Begin by importing the necessary libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
from mnist import MNIST
```

Load MNIST data as follows:

```
mndata = MNIST('.')
images, labels = mndata.load_training()
images_test, labels_test = mndata.load_testing()
images = np.array(images)  # Images are 28x28, already flattened to 784
images_test = np.array(images_test)
labels = np.array(labels)  # Convert to numpy array
labels_test = np.array(labels_test)
```

Print the shapes of the data:

```
print(f"images.shape: {images.shape}")
print(f"labels.shape: {labels.shape}")
print(f"images_test.shape: {images_test.shape}")
print(f"labels_test.shape: {labels_test.shape}")
```

```
images.shape: (60000, 784)
labels.shape: (60000,)
images_test.shape: (10000, 784)
labels_test.shape: (10000,)
```

Due to limited computing resources available to me, select a random subset of data as follows:

```
n_samples = 5000
np.random.seed(4)  # Set random seed for reproducibility
random_indices = np.random.choice(images.shape[0], n_samples, replace=False)
images = images[random_indices]
```

```
labels = labels[random_indices]
print(f"Subset images.shape: {images.shape}")
print(f"Subset labels.shape: {labels.shape}")
```

```
Subset images.shape: (5000, 784)
Subset labels.shape: (5000,)
```

One-hot encode the labels as follows:

```
labels = np.eye(10)[labels]
labels_test = np.eye(10)[labels_test]
print(f"labels.shape: {labels.shape}")
print(f"Head of labels (check 1-hot encoding): {labels[:5]}")
```

```
labels.shape: (5000, 10)
Head of labels (check 1-hot encoding): [[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

The problem doesn't specify the specific objective function to use, so we choose lasso. Define loss and lasso objective function as follows:

```
def loss(y, x, beta):
    return cp.sum_squares(y - x @ beta)/y.shape[0]

def l1_regularization(beta, lambda1):
    return lambda1 * cp.norm1(beta)

def lasso_objective(y, x, beta, lambda1):
    return loss(y, x, beta) + l1_regularization(beta, lambda1)
```

Create and solve the optimization problem as follows:

```
n = images.shape[0]
d = images.shape[1]
beta = cp.Variable((d, 10))
lambda1 = 3.0
y = labels  # One-hot encoded labels (labeled output data)
x = images  # Images (input data)
objective = lasso_objective(y, x, beta, lambda1)
problem = cp.Problem(cp.Minimize(objective))
problem.solve(solver=cp.CLARABEL)  # Use (default) CLARABEL solver
print(f"Optimal beta: {beta.value}")
```

```
Optimal beta: [[1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
```

```
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 ...
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]]
```

Calculate accuracy on test set. First, define a function to decode one-hot encoded labels:

```
def decode_one_hot(one_hot, axis=-1):
    return np.argmax(one_hot, axis=axis)
```

Calculate accuracy on test set as follows:

```
y_test = labels_test
x_test = images_test
predictions = x_test @ beta.value
print(f"predictions.shape: {predictions.shape}")
accuracy = np.mean(decode_one_hot(predictions) == decode_one_hot(y_test))
print(f"Test accuracy: {100*accuracy:.3g} percent")
```

```
predictions.shape: (10000, 10)
Test accuracy: 80.4 percent
```

Print the first 10 predictions and true labels:

```
print("First 10 predictions:")
print(decode_one_hot(predictions[:10], axis=1))
print("True labels:")
print(decode_one_hot(y_test[:10], axis=1))
```

```
First 10 predictions:
[7 2 1 0 4 1 4 9 5 7]
True labels:
[7 2 1 0 4 1 4 9 5 9]
```

Decode beta values to images and plot the matrices. First, define a function to decode beta values:

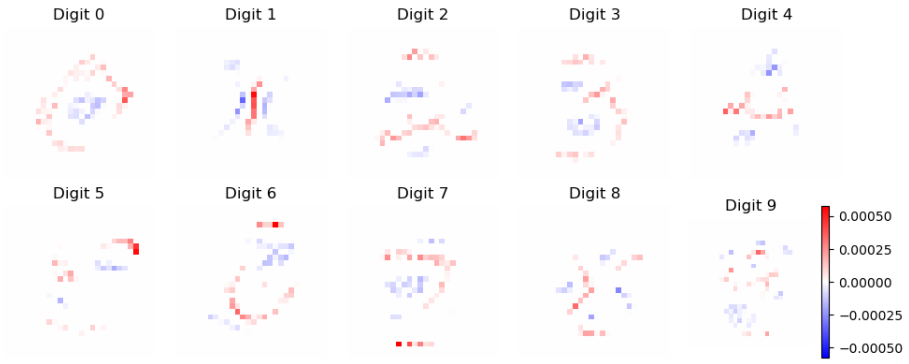
```
def decode_beta(beta):
    return beta.reshape(28, 28, 10)
```

Decode beta values:

```
beta_images = decode_beta(beta.value)
```

Plot the beta images:

```
fig, ax = plt.subplots(2, 5, figsize=(10, 4))
max_val = np.max(np.abs(beta_images))
min_val = -max_val
for i in range(10):
    if i != 9:
        ax[i//5, i%5].imshow(
            beta_images[:, :, i], vmin=min_val, vmax=max_val, cmap='bwr'
        )
    else:
        plt.colorbar(
            ax[i//5, i%5].imshow(
                beta_images[:, :, i], vmin=min_val, vmax=max_val,
                cmap='bwr'
            ),
            ax=ax[i//5, i%5]
        )
    ax[i//5, i%5].axis('off')
    ax[i//5, i%5].set_title(f"Digit {i}")
plt.tight_layout()
plt.show()
```




The above images show the learned weights for each digit (0-9) in the MNIST dataset. Interestingly, the weights appear to be somewhat interpretable, with the weights for each digit resembling the digit itself.