Sui_Wenyu_HW5_report

In [1]:
```
# import packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from IPython.display import display
import scipy.io as sio
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report,confusion_matrix
import matplotlib.pyplot as plt
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV, Ridge
from sklearn.model_selection import GridSearchCV, KFold
from scipy.stats import norm

import functions
```

1. Comparing multi-class classifiers for handwritten digits classification

1.1

The confusion matrix, precision, recall, and F-1 score for each of the classifiers are reported in the following code chunks.

In [2]:
```
# read data
matFile = sio.loadmat('./data/mnist_10digits')

xtrain = matFile['xtrain']
ytrain = matFile['ytrain'].flatten()
xtest = matFile['xtest']
ytest = matFile['ytest'].flatten()

# standardize data
xtrain = xtrain / 255
xtest = xtest / 255
```

In [4]:
```
# KNN, k=3
# We tried each integer between 2 to 10. Found that k=3 generates the highest overall accuracy measured by the training data.

KNN = KNeighborsClassifier(n_neighbors = 3).fit(xtrain, ytrain)
pred = KNN.predict(xtest)

functions.report_classification_result("KNN", ytest, pred)
```

```
Confusion Matrix of  KNN :
```

	0	1	2	3	4	5	6	7	8	9
0	974	1	1	0	0	1	2	1	0	0
1	0	1133	2	0	0	0	0	0	0	0
2	10	9	996	2	0	0	0	13	2	0
3	0	2	4	976	1	13	1	7	3	3
4	1	6	0	0	950	0	4	2	0	19
5	6	1	0	11	2	859	5	1	3	4
6	5	3	0	0	3	3	944	0	0	0
7	0	21	5	0	1	0	0	991	0	10
8	8	2	4	16	8	11	3	4	914	4
9	4	5	2	8	9	2	1	8	2	968

```
Precision, Recall, and F-1 score of  KNN :

              precision    recall  f1-score   support

           0     0.9663    0.9939    0.9799       980
           1     0.9577    0.9982    0.9776      1135
           2     0.9822    0.9651    0.9736      1032
           3     0.9635    0.9663    0.9649      1010
           4     0.9754    0.9674    0.9714       982
           5     0.9663    0.9630    0.9646       892
           6     0.9833    0.9854    0.9844       958
           7     0.9649    0.9640    0.9645      1028
           8     0.9892    0.9384    0.9631       974
           9     0.9603    0.9594    0.9598      1009

    accuracy                         0.9705     10000
   macro avg     0.9709    0.9701    0.9704     10000
weighted avg     0.9707    0.9705    0.9705     10000
```

In [5]:
```
# Logistic Regression
logistic_regression = LogisticRegression(max_iter = 1000).fit(xtrain, ytrain)
pred = logistic_regression.predict(xtest)

functions.report_classification_result("Logistic Regression", ytest, pred)
```

```
Confusion Matrix of  Logistic Regression :
```

	0	1	2	3	4	5	6	7	8	9
0	955	0	2	4	1	10	4	3	1	0
1	0	1110	5	2	0	2	3	2	11	0
2	6	9	930	14	10	3	12	10	34	4
3	4	1	16	925	1	23	2	10	19	9
4	1	3	7	3	921	0	6	5	6	30
5	9	2	3	35	10	777	15	6	31	4
6	8	3	8	2	6	16	912	2	1	0
7	1	7	23	7	6	1	0	947	4	32
8	9	11	6	22	7	29	13	10	855	12
9	9	8	1	9	21	7	0	21	9	924

```
Precision, Recall, and F-1 score of  Logistic Regression :

              precision    recall  f1-score   support

           0     0.9531    0.9745    0.9637       980
           1     0.9619    0.9780    0.9699      1135
           2     0.9291    0.9012    0.9149      1032
           3     0.9042    0.9158    0.9100      1010
           4     0.9369    0.9379    0.9374       982
           5     0.8952    0.8711    0.8830       892
           6     0.9431    0.9520    0.9475       958
           7     0.9321    0.9212    0.9266      1028
           8     0.8805    0.8778    0.8792       974
           9     0.9103    0.9158    0.9130      1009

    accuracy                         0.9256     10000
   macro avg     0.9246    0.9245    0.9245     10000
weighted avg     0.9254    0.9256    0.9254     10000
```

In [6]:
```
# SVM
SVM = SVC(kernel = 'linear').fit(xtrain, ytrain)
pred = SVM.predict(xtest)

functions.report_classification_result("SVM", ytest, pred)
```

```
Confusion Matrix of  SVM :
```

	0	1	2	3	4	5	6	7	8	9
0	957	0	4	1	1	6	9	1	0	1
1	0	1122	3	2	0	1	2	1	4	0
2	8	6	967	11	3	3	7	8	17	2
3	4	3	16	947	1	16	0	9	12	2
4	1	1	10	1	942	2	4	2	3	16
5	10	4	3	36	6	803	13	1	14	2
6	9	2	13	1	5	16	910	1	1	0
7	1	8	21	10	8	1	0	957	3	19
8	8	4	6	25	7	26	6	7	877	8
9	7	7	2	11	33	4	0	18	5	922

```
Precision, Recall, and F-1 score of  SVM :

              precision    recall  f1-score   support

           0     0.9522    0.9765    0.9642       980
           1     0.9697    0.9885    0.9791      1135
           2     0.9254    0.9370    0.9312      1032
           3     0.9062    0.9376    0.9217      1010
           4     0.9364    0.9593    0.9477       982
           5     0.9146    0.9002    0.9073       892
           6     0.9569    0.9499    0.9534       958
           7     0.9522    0.9309    0.9415      1028
           8     0.9370    0.9004    0.9183       974
           9     0.9486    0.9138    0.9308      1009

    accuracy                         0.9404     10000
   macro avg     0.9399    0.9394    0.9395     10000
weighted avg     0.9405    0.9404    0.9403     10000
```

In [7]:
```
# kernel SVM
kernel_SVM = SVC(kernel = 'rbf').fit(xtrain, ytrain)
pred = kernel_SVM.predict(xtest)

functions.report_classification_result("Kernel SVM", ytest, pred)
```

```
Confusion Matrix of  Kernel SVM :
```

	0	1	2	3	4	5	6	7	8	9
0	973	0	1	0	0	2	1	1	2	0
1	0	1126	3	1	0	1	1	1	2	0
2	6	1	1006	2	1	0	2	7	6	1
3	0	0	2	995	0	2	0	5	5	1
4	0	0	5	0	961	0	3	0	2	11
5	2	0	0	9	0	871	4	1	4	1
6	6	2	0	0	2	3	944	0	1	0
7	0	6	11	1	1	0	0	996	2	11
8	3	0	2	6	3	2	2	3	950	3
9	3	4	1	7	10	2	1	7	4	970

```
Precision, Recall, and F-1 score of  Kernel SVM :

              precision    recall  f1-score   support

           0     0.9799    0.9929    0.9863       980
           1     0.9886    0.9921    0.9903      1135
           2     0.9758    0.9748    0.9753      1032
           3     0.9745    0.9851    0.9798      1010
           4     0.9826    0.9786    0.9806       982
           5     0.9864    0.9765    0.9814       892
           6     0.9854    0.9854    0.9854       958
           7     0.9755    0.9689    0.9722      1028
           8     0.9714    0.9754    0.9734       974
           9     0.9719    0.9613    0.9666      1009

    accuracy                         0.9792     10000
   macro avg     0.9792    0.9791    0.9791     10000
weighted avg     0.9792    0.9792    0.9792     10000
```

In [9]:
```
# neural network, hidden layer sizes = (20, 10)
neural_network = MLPClassifier(hidden_layer_sizes = (20, 10), max_iter = 1000).fit(xtrain, ytrain)
pred = neural_network.predict(xtest)

functions.report_classification_result("Neural Network", ytest, pred)
```

```
Confusion Matrix of  Neural Network :
```

	0	1	2	3	4	5	6	7	8	9
0	956	0	4	2	3	2	6	1	4	2
1	0	1117	4	3	0	2	2	0	7	0
2	7	5	968	20	5	0	3	14	10	0
3	1	2	10	969	2	12	0	5	7	2
4	3	1	6	2	929	0	6	3	4	28
5	8	1	0	24	4	838	10	0	5	2
6	10	3	5	1	4	13	920	0	1	1
7	2	4	11	20	7	2	0	955	3	24
8	7	3	5	23	7	13	8	4	891	13
9	0	2	0	11	18	7	0	7	9	955

```
Precision, Recall, and F-1 score of  Neural Network :

              precision    recall  f1-score   support

           0     0.9618    0.9755    0.9686       980
           1     0.9815    0.9841    0.9828      1135
           2     0.9556    0.9380    0.9467      1032
           3     0.9014    0.9594    0.9295      1010
           4     0.9489    0.9460    0.9475       982
           5     0.9426    0.9395    0.9410       892
           6     0.9634    0.9603    0.9618       958
           7     0.9656    0.9290    0.9470      1028
           8     0.9469    0.9148    0.9305       974
           9     0.9299    0.9465    0.9381      1009

    accuracy                         0.9498     10000
   macro avg     0.9498    0.9493    0.9494     10000
weighted avg     0.9502    0.9498    0.9498     10000
```

1.2

According the results above, the performance of the classifiers are ranked as below.
(Performance is measured by macro average F1 score, ranked from highest to lowest.)

1. Kernel SVM (macro avg F1 score: 0.98)
2. KNN (macro avg F1 score: 0.97)
3. Neural Network (macro avg F1 score: 0.95)
4. SVM (macro avg F1 score: 0.94)
5. Logistic Regression (macro avg F1 score: 0.92)

Comments:

1. Linear SVM and Logistic Regression are the two models performing worst. These two models make classifications using linear decision boundaries, while the other three models make classifications using non-linear decision boundaries. Therefore, it is possible that the linear SVM model and the logistic regression model perform worse than the other three models becuase non-linear decision boundary is more suitable for this

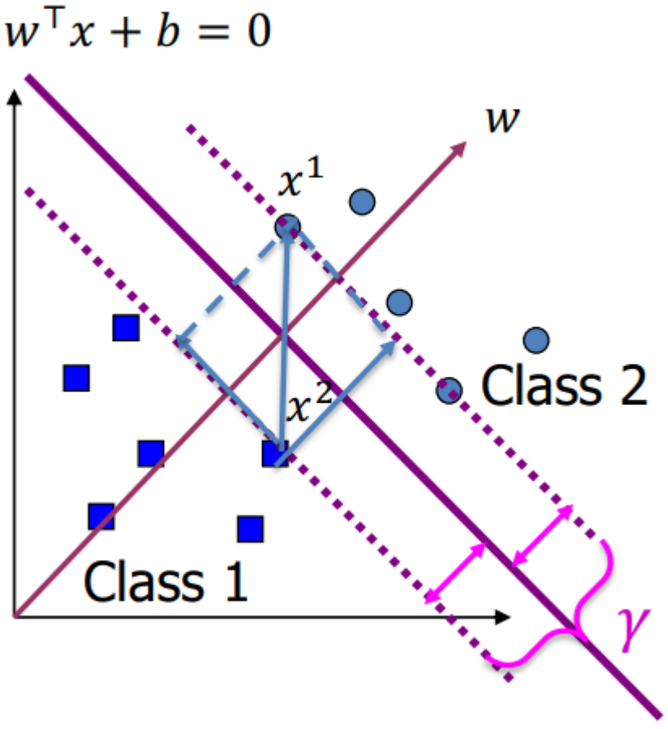
classification problem.

2. In this problem, we didn't perform any preprocessing steps to the data. It is possible that some predictor variables are correlated to each other in the dataset. The performance of the logistic regression model may be affected by the issue of multicolinearity. That potentially explains why it performs worst among all the models.

3. Neural network performs worse than the kernel SVM model and the KNN model. It is possibly because the structure of the neural network is too simple, or becuase some hyperparameters are not tuned in the best way. The performance of neural network could be improved after the network is redesigned and the hyperparameters are tuned.

2. SVM

2.1



The image above shows an example of SVM model, where the decision boundary is represented by $w^T x + b = 0$, and the distance between the two dash lines represents the margin. w is a vector which is orthogonal to the decision boundary.

For all x in Class 2 ($y = 1$), we have $w^T x + b \geq c$
For all x in Class 1 ($y = -1$), we have $w^T x + b \leq -c$
These can also be written as $(w^T + b)y \geq c$

If we pick two data points x_1 and x_2, which are located on each dash line respectively, we can get

$$Unnormalized\ Margin = w^T(x_1 - x_2) = 2c$$
$$Normalized\ Margin\ (\gamma) = \frac{w^T(x_1 - x_2)}{||w||} = \frac{2c}{||w||}$$

Since we want to maximize the margin, we have the following optimization problem:

$$max_{w,b}\ \gamma = \frac{2c}{||w||}$$
$$s.t.\ \ y^i(w^Tx^i + b) \geq c, \forall i$$

In this optimization problem, the magnitude of c merely scales w and b, and does not change the optimization solution. Therefore, we can set $c = 1$ to get a cleaner problem:

$$max_{w,b}\ \gamma = \frac{2}{||w||}$$
$$s.t.\ \ y^i(w^Tx^i + b) \geq 1, \forall i$$

2.2

The optimization problem mentioned in the last question can also be converted into:

$$min_{w,b}\ \frac{1}{2}w^Tw$$
$$s.t.\ \ 1 - y^i(w^Tx^i + b) \leq 0, \forall i$$

The Lagrangian function can be written as:

$$L(w, b, \alpha) = \frac{1}{2}w^Tw + \sum_{i=1}^{m}\alpha_i(1 - y^i(w^Tx^i + b))$$

where $\alpha_i \geq 0$ is the Lagrangian multiplier.

Taking derivative of the function above and setting to zero, we have:

$$\frac{\partial L(w, b, \alpha)}{\partial w} = w - \sum_{i=1}^{m}\alpha_i y^i x^i = 0$$

Therefore, $w = \sum_{i=1}^{m}\alpha_i y^i x^i$

Since $y^i = \pm 1$, and $\alpha_i \geq 0$, it can be seen that the optimal weight vector w is a linear combination of the feature vectors x^i from the data.

2.3

Here's the Lagrangian function derived in the last question:

$$L(w, b, \alpha) = \frac{1}{2}w^Tw + \sum_{i=1}^{m}\alpha_i(1 - y^i(w^Tx^i + b))$$

if there exists some saddle point of L, then the saddle point should satisfy the following KKT condition:

$$\alpha_i(1 - y^i(w^Tx^i + b)) = 0, \forall i$$

In other words, for data points with $(1 - y^i(w^T x^i + b)) < 0$, we have $\alpha_i = 0$
For data points with $(1 - y^i(w^T x^i + b)) = 0$, we have $\alpha_i \geq 0$

Since $w = \sum_{i=1}^{m} \alpha_i y^i x^i$, we can know that only data points with $(1 - y^i(w^T x^i + b)) = 0$ will contribute to w. ($\alpha_i \neq 0$)
In other words, only the data points on the "margin" will contribute to w.

2.4

2.4(a)

The four training data points x_1, x_2, x_3, x_4 are displayed in the plots generated by the code chunk below. Here, we define that the horizontal axis represents variable x^1 and the vertical axis represents the variable x^2,

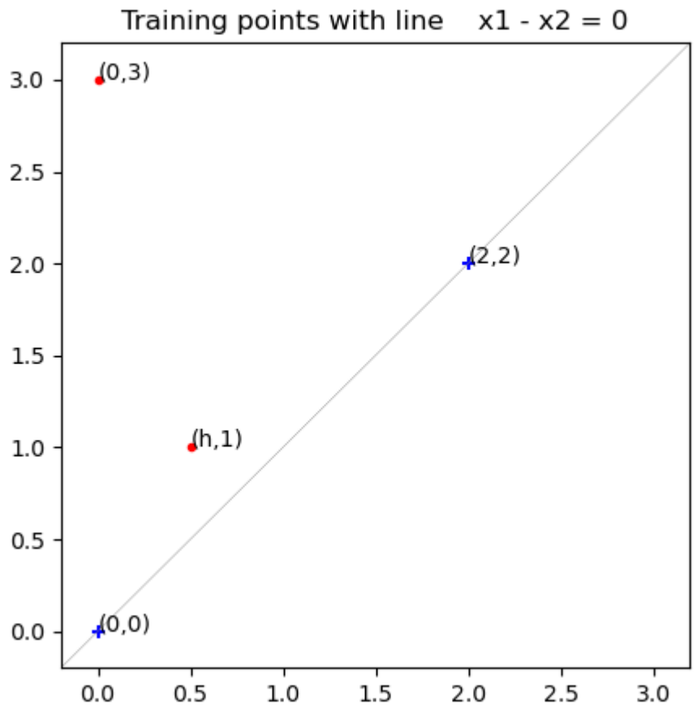
In the first graph, the gray line going through both $x_1(0,0)$ and $x_2(2,2)$ can be written as $x^1 - x^2 = 0$
In this situation, we can easily see that the training points are linearly separatble if $x_3(h,1)$ is above the gray line.
Since we know that $h > 0$, we can conclude that the training data points are linearly separable if $0 < h < 1$.

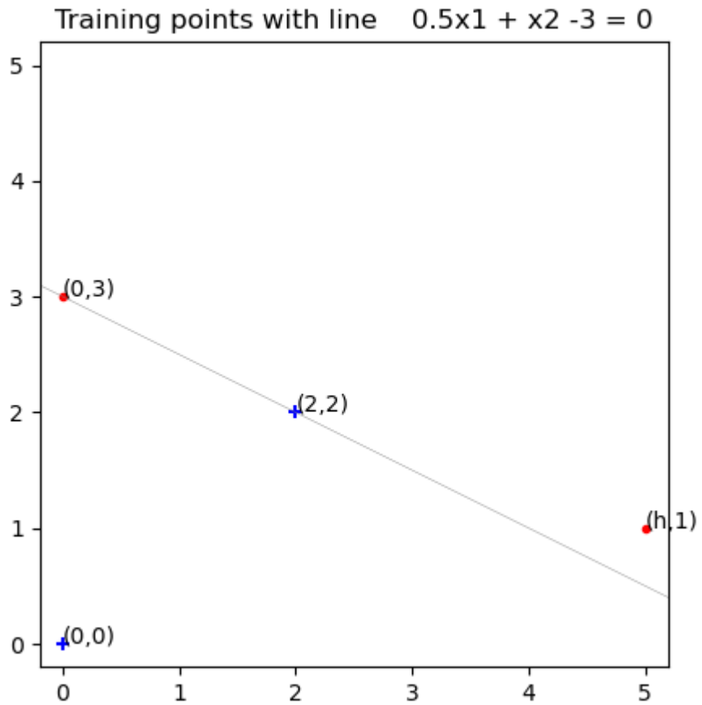
The second graph displays a different situation. The straight line going through both $x_4(0,3)$ and $x_2(2,2)$ can be written as $0.5x^1 + x^2 - 3 = 0$
It can be seen that the training data points are linearly seperable if $x_3(h,1)$ is above this line.
Therefore, we can find out that the training data points are linearly separable if $h > 4$

To conclude, the training data points are linearly separable if $0 < h < 1$ or $h > 4$

In [4]:
```
# Situation 1
plt.figure (figsize=(5,5))
plt.scatter(0, 3, c='r', marker = '.')
plt.annotate(text = '(0,3)', xy = (0,3))

plt.scatter(0.5, 1, c='r', marker = '.')
plt.annotate(text = '(h,1)', xy = (0.5,1))

plt.scatter(0, 0, c='b', marker = '+')
plt.annotate(text = '(0,0)', xy = (0,0))

plt.scatter(2, 2, c='b', marker = '+')
plt.annotate(text = '(2,2)', xy = (2,2))

plt.axline((0, 0), (1, 1), linewidth=0.3, color='gray')

plt.xlim(-0.2,3.2)
plt.ylim(-0.2,3.2)
plt.title("Training points with line    x1 - x2 = 0")
plt.show()


# Situation 2
plt.figure (figsize=(5,5))
plt.scatter(0, 3, c='r', marker = '.')
plt.annotate(text = '(0,3)', xy = (0,3))

plt.scatter(5, 1, c='r', marker = '.')
plt.annotate(text = '(h,1)', xy = (5,1))

plt.scatter(0, 0, c='b', marker = '+')
plt.annotate(text = '(0,0)', xy = (0,0))
```

```
plt.scatter(2, 2, c='b', marker = '+')
plt.annotate(text = '(2,2)', xy = (2,2))

plt.axline((0, 3), (2, 2), linewidth=0.3, color='gray')

plt.xlim(-0.2,5.2)
plt.ylim(-0.2,5.2)
plt.title("Training points with line    0.5x1 + x2 -3 = 0")
plt.show()
```





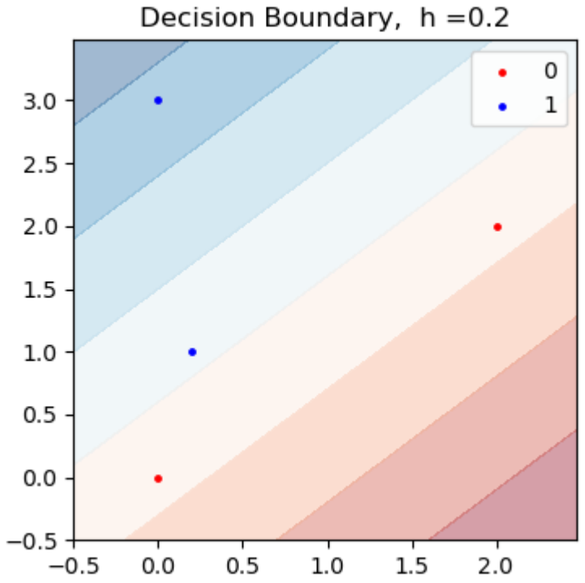
2.4(b)

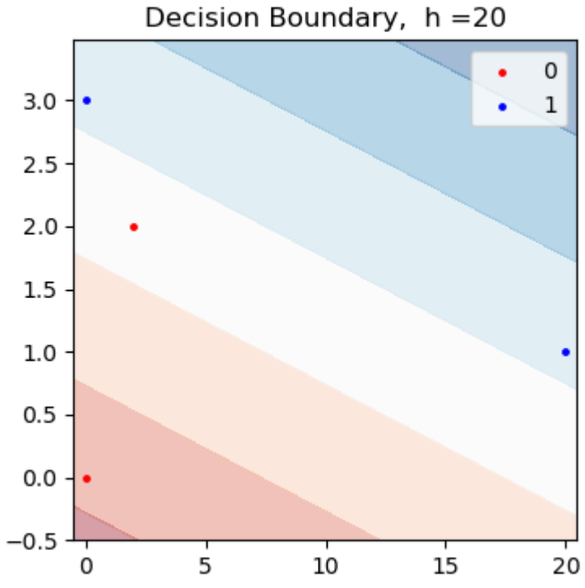
The orientation of the maximum margin decision boundary will change as h changes, when the points are separable.

The first graph generated by the following code chunk shows the orientation of the maximum margin decision boundary when $h = 0.2$.
The second graph shows the orientation of the maximum margin decision boundary when $h = 20$.

It can be clearly seen that the orientations of the decision boundary are different in these two graphs.

In [5]:
```
# h = 0.2
x = np.array([[0,0],
                        [2,2],
                        [0.2,1],
                        [0,3]])
y = np.array([0,0,1,1])

clf = SVC(kernel = 'linear')

functions.show_decision_boundary(clf, x, y, 0.2)


# h = 8
x = np.array([[0,0],
                        [2,2],
                        [20,1],
                        [0,3]])
y = np.array([0,0,1,1])

clf = SVC(kernel = 'linear')

functions.show_decision_boundary(clf, x, y, 20)
```





3.Neural networks and backward propagation

3.1

Since $L(w, \alpha, \beta) = \sum_{i=1}^{m} (y^i - \sigma(w^T z^i))^2$,

we can define $L_i(w, \alpha, \beta) = (y^i - \sigma(w^T z^i))^2$

Therefore, $L(w, \alpha, \beta) = \sum_{i=1}^{m} L_i(w, \alpha, \beta)$

Also, we can get

$$\begin{aligned}\frac{\partial L(w,\alpha,\beta)}{\partial w} &= \sum_{i=1}^{m} \frac{\partial L_i(w,\alpha,\beta)}{\partial w} \\ &= \sum_{i=1}^{m} \frac{\partial L_i(w,\alpha,\beta)}{\partial \sigma(u^i)} \frac{\partial \sigma(u^i)}{u^i} \frac{\partial u^i}{w}\end{aligned} \tag{1}$$

Then we can derive

$$\begin{aligned}\frac{\partial L_i(w,\alpha,\beta)}{\partial \sigma(u^i)} &= \frac{\partial (y^i - \sigma(u^i))^2}{\partial \sigma(u^i)} \\ &= -2(y^i - \sigma(u^i))\end{aligned} \tag{2}$$

Also,

$$\begin{aligned}\frac{\partial \sigma(u^i)}{\partial u^i} &= \frac{\partial (1+e^{-u^i})^{-1}}{\partial u^i} \\ &= -(1+e^{-u^i})^{-2} \times e^{-u^i} \times (-1) \\ &= \frac{1}{1+e^{-u^i}} \times \frac{e^{-u^i}}{1+e^{-u^i}} \\ &= \sigma(u^i)(1-\sigma(u^i))\end{aligned} \tag{3}$$

And,

$$\frac{\partial u^i}{\partial w} = \frac{\partial w^T z^i}{\partial w} = z^i \tag{4}$$

Therefore, we have

$$\begin{aligned}\frac{\partial L(w,\alpha,\beta)}{\partial w} &= \sum_{i=1}^{m} \frac{\partial L_i(w,\alpha,\beta)}{\partial \sigma(u^i)} \frac{\partial \sigma(u^i)}{\partial u^i} \frac{\partial u^i}{\partial w} \\ &= \sum_{i=1}^{m} -2(y^i - \sigma(u^i))\sigma(u^i)(1-\sigma(u^i))z^i \\ &= -\sum_{i=1}^{m} 2(y^i - \sigma(u^i))\sigma(u^i)(1-\sigma(u^i))z^i\end{aligned} \tag{5}$$

3.2

To calculate gradient of $L(w,\alpha,\beta)$ as of α :

Using similar approach as last question, we can derive

$$\begin{aligned}\frac{\partial L_i(w,\alpha,\beta)}{\partial z_1^i} &= \frac{\partial L_i(w,\alpha,\beta)}{\partial \sigma(u^i)} \frac{\partial \sigma(u^i)}{\partial u^i} \frac{\partial u^i}{\partial z_1^i} \\ &= -2(y^i - \sigma(u^i))\sigma(u^i)(1-\sigma(u^i))w_1\end{aligned} \tag{6}$$

Let $v_1^i = \alpha^T x^i$, we have

$$
\begin{aligned}
\frac{\partial z_1^i}{\partial \alpha} &= \frac{\partial z_1^i}{\partial v_1^i}\frac{\partial v_1^i}{\partial \alpha} \\
&= \frac{\partial (1+e^{-v_1^i})^{-1}}{\partial v_1^i}x^i \\
&= -(1+e^{-v_1^i})^{-2}\cdot e^{-v_1^i}\cdot(-1)\cdot x^i \\
&= \frac{1}{1+e^{-v_1^i}}\cdot\frac{e^{-v_1^i}}{1+e^{-v_1^i}}\cdot x^i \\
&= \sigma(v_1^i)(1-\sigma(v_1^i))x^i
\end{aligned} \tag{7}
$$

Therefore,

$$
\begin{aligned}
\frac{\partial L_i(w,\alpha,\beta)}{\partial \alpha} &= \frac{\partial L_i(w,\alpha,\beta)}{\partial z_1^i}\frac{\partial z_1^i}{\partial \alpha} \\
&= -2(y^i-\sigma(u^i))\sigma(u^i)(1-\sigma(u^i))w_1\sigma(v_1^i)(1-\sigma(v_1^i))x^i
\end{aligned} \tag{8}
$$

So,

$$
\begin{aligned}
\frac{\partial L(w,\alpha,\beta)}{\partial \alpha} &= \sum_{i=1}^{m}\frac{\partial L_i(w,\alpha,\beta)}{\partial \alpha} \\
&= -\sum_{i=1}^{m}2(y^i-\sigma(u^i))\sigma(u^i)(1-\sigma(u^i))w_1\sigma(v_1^i)(1-\sigma(v_1^i))x^i
\end{aligned} \tag{9}
$$

To calculate gradient of $L(w,\alpha,\beta)$ as of β :

Using similar approach, we can get

$$
\begin{aligned}
\frac{\partial L_i(w,\alpha,\beta)}{\partial z_2^i} &= \frac{\partial L_i(w,\alpha,\beta)}{\partial \sigma(u^i)}\frac{\partial \sigma(u^i)}{\partial u^i}\frac{\partial u^i}{\partial z_2^i} \\
&= -2(y^i-\sigma(u^i))\sigma(u^i)(1-\sigma(u^i))w_2
\end{aligned} \tag{10}
$$

Let $v_2^i = \beta^T x^i$, we have

$$
\begin{aligned}
\frac{\partial z_2^i}{\partial \beta} &= \frac{\partial z_2^i}{\partial v_2^i}\frac{\partial v_2^i}{\partial \beta} \\
&= \frac{\partial (1+e^{-v_2^i})^{-1}}{\partial v_2^i}x^i \\
&= -(1+e^{-v_2^i})^{-2}\cdot e^{-v_2^i}\cdot(-1)\cdot x^i \\
&= \frac{1}{1+e^{-v_2^i}}\cdot\frac{e^{-v_2^i}}{1+e^{-v_2^i}}\cdot x^i \\
&= \sigma(v_2^i)(1-\sigma(v_2^i))x^i
\end{aligned} \tag{11}
$$

Therefore,

$$\begin{aligned}\frac{\partial L_i(w,\alpha,\beta)}{\partial \beta} &= \frac{\partial L_i(w,\alpha,\beta)}{\partial z_2^i}\frac{\partial z_2^i}{\partial \beta} \\ &= -2(y^i-\sigma(u^i))\sigma(u^i)(1-\sigma(u^i))w_2\sigma(v_2^i)(1-\sigma(v_2^i))x^i\end{aligned} \tag{12}$$

So,

$$\begin{aligned}\frac{\partial L(w,\alpha,\beta)}{\partial \beta} &= \sum_{i=1}^{m}\frac{\partial L_i(w,\alpha,\beta)}{\partial \beta} \\ &= -\sum_{i=1}^{m} 2(y^i-\sigma(u^i))\sigma(u^i)(1-\sigma(u^i))w_2\sigma(v_2^i)(1-\sigma(v_2^i))x^i\end{aligned} \tag{13}$$

4. Feature selection and change-point detection

4.1

The mutual information of $spam$ and $prize$ can be calculated as follows:

$$\begin{aligned}I(spam, prize) &= \sum_{i=0}^{1}\sum_{j=0}^{1} P(prize=i, spam=j)\, log_2\frac{P(prize=i,spam=j)}{P(prize=i)P(spam=j)} \\ &= \frac{150}{16160}\, log_2\frac{150\times 16160}{(150+1000)\times(150+10)} + \frac{10}{16160}\, log_2\frac{10\times 16160}{(10+15000)\times(150+10)} + \frac{1000}{16160}\, log_2\frac{1000\times 16160}{(150+1000)\times(1000+15000)} + \frac{15000}{16160}\, log_2\frac{15000\times 16160}{(10+15000)\times(1000+15000)} \\ &\approx 0.03296011876395398\end{aligned} \tag{14}$$

The mutual information of $spam$ and $hello$ can be calculated as follows:

$$\begin{aligned}I(spam, hello) &= \sum_{i=0}^{1}\sum_{j=0}^{1} P(prize=i, hello=j)\, log_2\frac{P(prize=i,spam=j)}{P(prize=i)P(spam=j)} \\ &= \frac{145}{16160}\, log_2\frac{145\times 16160}{(145+11000)\times(145+15)} + \frac{15}{16160}\, log_2\frac{15\times 16160}{(15+5000)\times(145+15)} + \frac{11000}{16160}\, log_2\frac{11000\times 16160}{(145+11000)\times(11000+5000)} + \frac{5000}{16160}\, log_2\frac{5000\times 16160}{(15+5000)\times(11000+5000)} \\ &\approx 0.0019480406287359392\end{aligned} \tag{15}$$

Since $I(spam, prize) > I(spam, hello)$, "**prize**" is is more informative for deciding whether or not the email is spam than "**hello**".

In [2]:
```
v11 = 150
v12 = 10
v21 = 1000
v22 = 15000

N = v11 + v12 + v21 + v22

Ispam_prize = v11/N * np.log2((v11/N) /((v11+ v21) / N) / ((v11 + v12) / N) ) \
                        + v12 / N *  np.log2((v12/N) / ((v12+ v22) / N) / ((v11 + v12) / N) ) \
                        + v21 / N *  np.log2((v21/N) / ((v11+ v21) / N) / ((v21 + v22) / N) ) \
                        + v22 / N *  np.log2((v22/N) / ((v12+ v22) / N) / ((v21 + v22) / N) )

print("I(spam, prize) = ", Ispam_prize )
```

```
I(spam, prize) =  0.03296011876395398
```

In [3]:
```
v11 = 145
v12 = 15
v21 = 11000
v22 = 5000

N = v11 + v12 + v21 + v22

Ispam_hello = v11/N * np.log2((v11/N) /((v11+ v21) / N) / ((v11 + v12) / N) ) \
                        + v12 / N *  np.log2((v12/N) / ((v12+ v22) / N) / ((v11 + v12) / N) ) \
                        + v21 / N *  np.log2((v21/N) / ((v11+ v21) / N) / ((v21 + v22) / N) ) \
                        + v22 / N *  np.log2((v22/N) / ((v12+ v22) / N) / ((v21 + v22) / N) )

print("I(spam, hello) = ", Ispam_hello )
```

```
I(spam, hello) =  0.0019480406287359392
```

4.2

Since $f_0 = N(0,1)$, its pdf can be written as

$$f_0(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

Similarly, since $f_0 = N(0.5,1.5)$, its pdf can be written as

$$f_1(x) = \frac{1}{\sqrt{1.5}\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-0.5}{\sqrt{1.5}})^2} = \frac{1}{\sqrt{3\pi}} e^{-\frac{(x-0.5)^2}{3}}$$

$$\begin{aligned} \frac{f_1(x)}{f_0(x)} &= \frac{\frac{1}{\sqrt{3\pi}} e^{-\frac{(x-0.5)^2}{3}}}{\frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}} \\ &= \sqrt{\frac{2}{3}} e^{\frac{1}{2}x^2 - \frac{(x-0.5)^2}{3}} \end{aligned} \tag{16}$$

Therefore,

$$log\frac{f_1(x)}{f_0(x)} = log\sqrt{\frac{2}{3}} + \frac{1}{2}x^2 - \frac{(x-0.5)^2}{3}$$

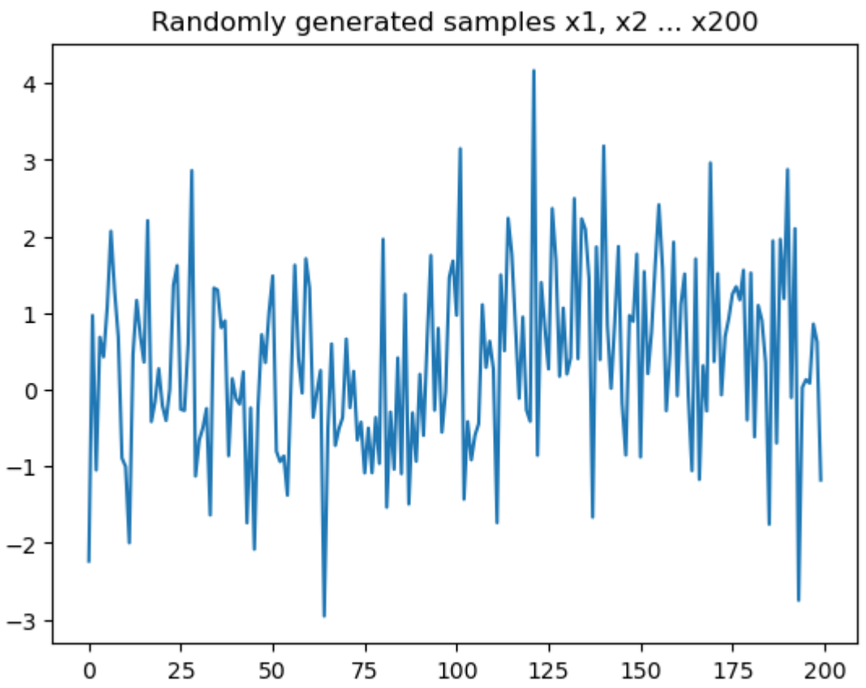
The CUSUM statistic can be written as:

$$W_0 = 0$$

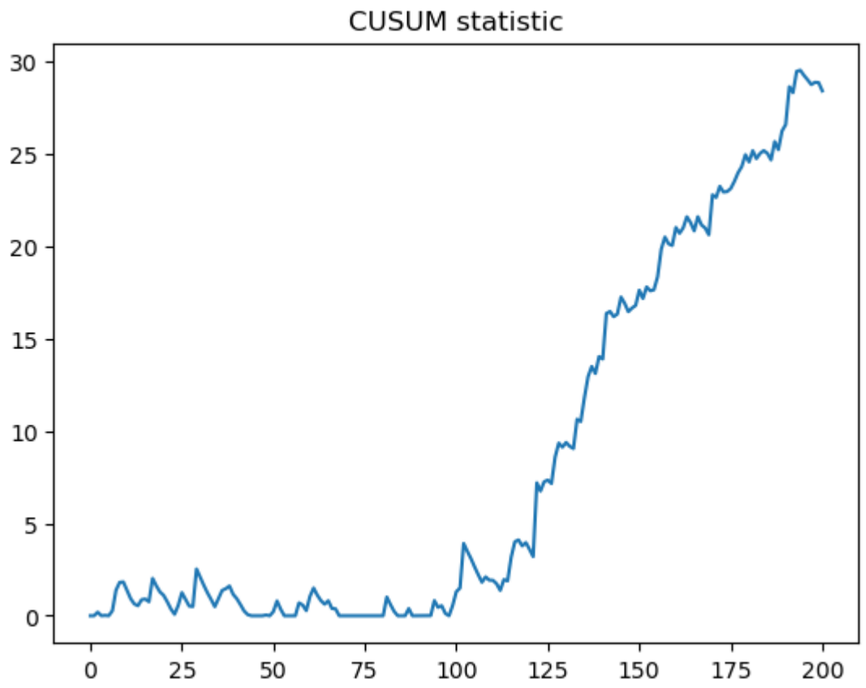
$$\begin{aligned} W_t &= max(W_{t-1} + log\frac{f_1(x)}{f_0(x)}, 0) \\ &= max(W_{t-1} + log\sqrt{\frac{2}{3}} + \frac{1}{2}x^2 - \frac{(x-0.5)^2}{3}, 0) \end{aligned} \tag{17}$$

The randomly generated samples $x_1, x_2 \dots x_{200}$ and their CUSUM statistics are displayed in the graphs below.

In [4]:
```
# generate samples
x1 = np.random.randn(100)
x2 =  0.5 + np.random.randn(100) * np.sqrt(1.5)

x = np.concatenate((x1, x2))

plt.figure()
plt.plot(x)
plt.title("Randomly generated samples x1, x2 ... x200")
plt.show()
```



In [5]:
```
# Calculate CUSUM
W = [0]
w = 0

for x_value in x:
    log_f0 = norm.logpdf(x_value, loc=0, scale=1)
    log_f1 = norm.logpdf(x_value, loc=0.5, scale= np.sqrt(1.5))

    if w + (log_f1 - log_f0) > 0:
        w = w + (log_f1 - log_f0)
    else:
        w = 0

    W += [w]

plt.figure()
plt.plot(W)
plt.title("CUSUM statistic")
plt.show()
```



5. Medical imaging reconstruction

5.1

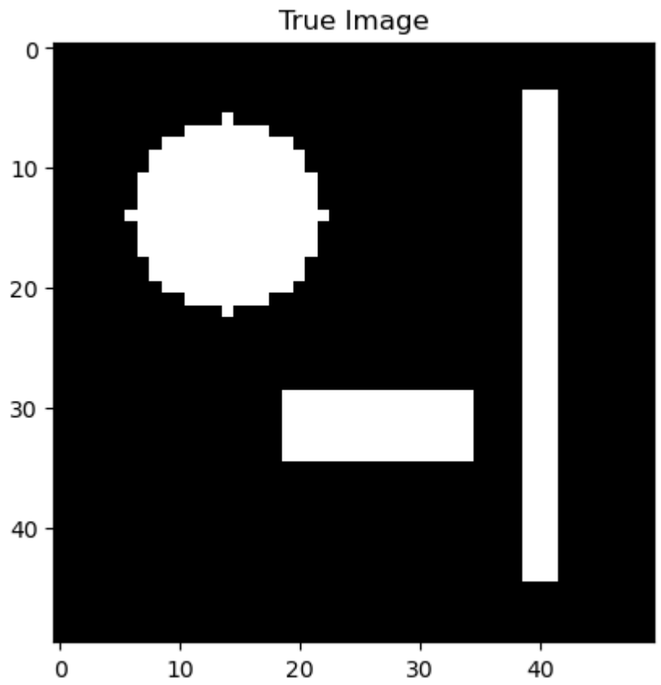
Here are the purposes of the following code chunks:

1. The first code chunk reads the data and shows the true image.

2. The second code chunk generates data of y, A and ϵ. It also performs 10-fold cross validation using LASSO regression, displays the cv error curve and the best λ value for LASSO regression. Since we know that the values of x is either 0 or 1, we restrict the regression coefficients to be positive. The cv error curve displays the average mean squared error for different λ values. The best λ is the λ value that generates the least average MSE in the 10 fold cross validation. The best λ approximately equals to 0.044. (This value may vary each time when the code runs due to randomness.)

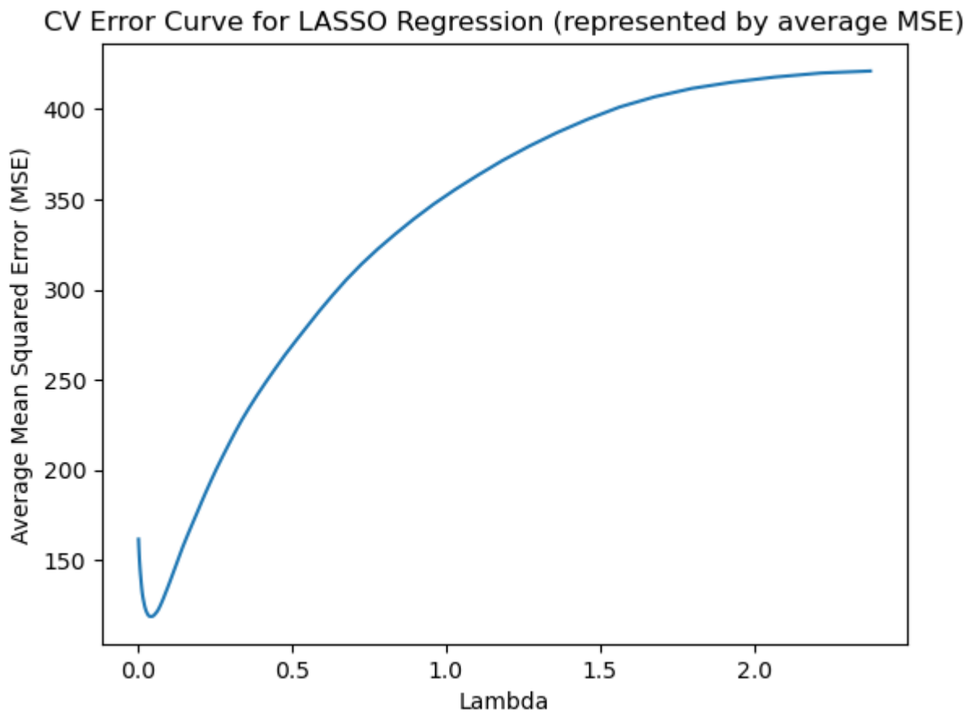
3. The third code chunk extracts the coefficients from the LASSO regression model selected by cross validation and recovers the image.

In [2]:
```
# read data
matFile = sio.loadmat('./data/cs.mat')
x = matFile['img']

# Show original image
plt.figure()
plt.imshow(x, cmap = 'gray')
plt.title("True Image")
plt.show()
```

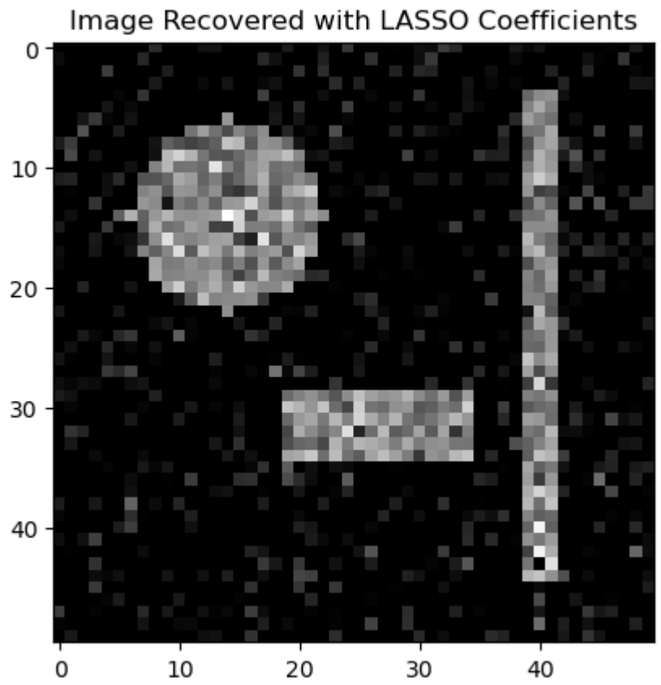


In [3]:
```
x= np.reshape(x, (2500,))

# generate matrix A, epsilon and y
A = np.random.randn(1300, 2500)
eps = np.random.randn(1300) * 5
y = np.dot(A, x) + eps

# LASSO cross validation
clf = LassoCV(cv = 10, positive = True).fit(A,y)

mse_path = clf.mse_path_
alphas = clf.alphas_
best_alpha = clf.alpha_
print("Best Lambda value chosen by 10 fold cross validation: ", round(best_alpha,5))

mean_mse_path = np.mean(mse_path, axis = 1)

# display cv error curve, represented by mean MSE
plt.figure()
plt.plot(alphas, mean_mse_path)
plt.xlabel("Lambda")
plt.ylabel("Average Mean Squared Error (MSE)")
plt.title("CV Error Curve for LASSO Regression (represented by average MSE)")
plt.show()
```

```
Best Lambda value chosen by 10 fold cross validation:  0.04452
```



In [4]:
```
# retrieve LASSO coefficient and reconstruct the image
importance = clf.coef_
reconstruct_lasso = np.reshape(importance, (50,50))

plt.figure()
plt.imshow(reconstruct_lasso, cmap = "gray")
plt.title("Image Recovered with LASSO Coefficients")
plt.show()
```



5.2

Here are the purposes of the following code chunks:

1. The first code chunk performs 10-fold cross validation using Ridge regression. It also displays the cv error curve and the best λ value. Since we know that the values of x is either 0 or 1, we restrict the regression coefficients to be positive. The cv error curve displays the average mean squared error for different λ values. The best λ is the λ value that generates the least average MSE in the 10 fold cross validation. The best λ approximately equals to 77. (This value may vary each time when the code runs due to randomness.)

2. The second code chunk extracts the coefficients from the Ridge regression model selected by cross validation and recovers the image.

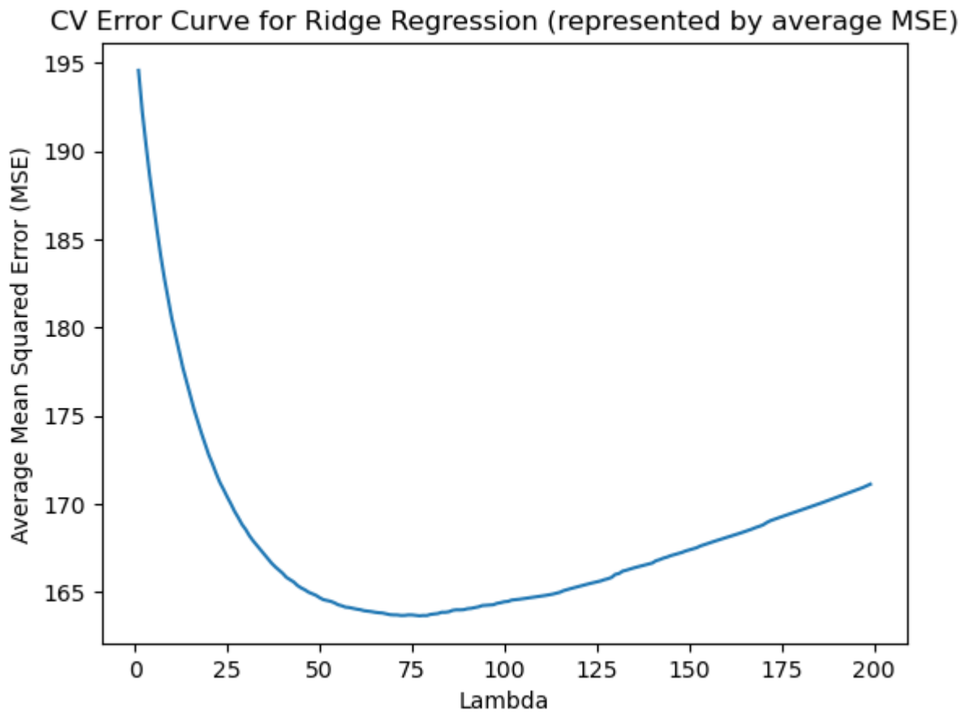
3. The third code chunk displays both the image recovered with LASSO regression and the image recovered with Ridge regression.

We can see that the LASSO regression gives a better recoved image, since the image recovered by LASSO regression has obviously fewer noises than the image recovered by Ridge regression.

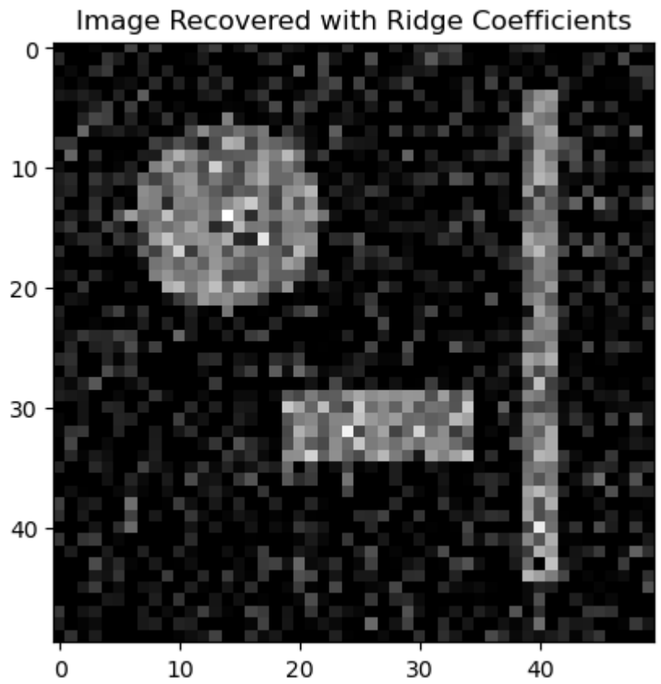
In [5]:
```
# Ridge cross validation
cv_split = KFold(n_splits = 10, shuffle = True)

ridge = Ridge(positive = True)
alphas = np.arange(1,200,1)
para = {'alpha': alphas}

cv_search = GridSearchCV(estimator = ridge,
                         scoring = 'neg_mean_squared_error',
                         cv = cv_split,
                         param_grid = para,
                         ).fit(A, y)

mean_mse_path = -cv_search.cv_results_['mean_test_score']
best_alpha = cv_search.best_params_['alpha']
print("Best Lambda value chosen by 10 fold cross validation: ", best_alpha)
```

```
plt.figure()
plt.plot(alphas, mean_mse_path)
plt.xlabel("Lambda")
plt.ylabel("Average Mean Squared Error (MSE)")
plt.title("CV Error Curve for Ridge Regression (represented by average MSE)")
plt.show()
```

```
Best Lambda value chosen by 10 fold cross validation:  77
```



In [6]:
```
model = Ridge(alpha = best_alpha, positive = True).fit(A,y)

coef = model.coef_
reconstruct_ridge = np.reshape(coef, (50,50))

plt.figure()
plt.imshow(reconstruct_ridge, cmap = "gray")
plt.title("Image Recovered with Ridge Coefficients")
plt.show()
```



In [7]:
```
# compare the images reconstructed form LASSO and Ridge
plt.figure(figsize = (10,20))
plt.subplot(1,2,1)
plt.imshow(reconstruct_lasso, cmap = "gray")
plt.title("Image Recovered with LASSO Coefficients")

plt.subplot(1,2,2)
plt.imshow(reconstruct_ridge, cmap = "gray")
plt.title("Image Recovered with Ridge Coefficients")
plt.show()
```

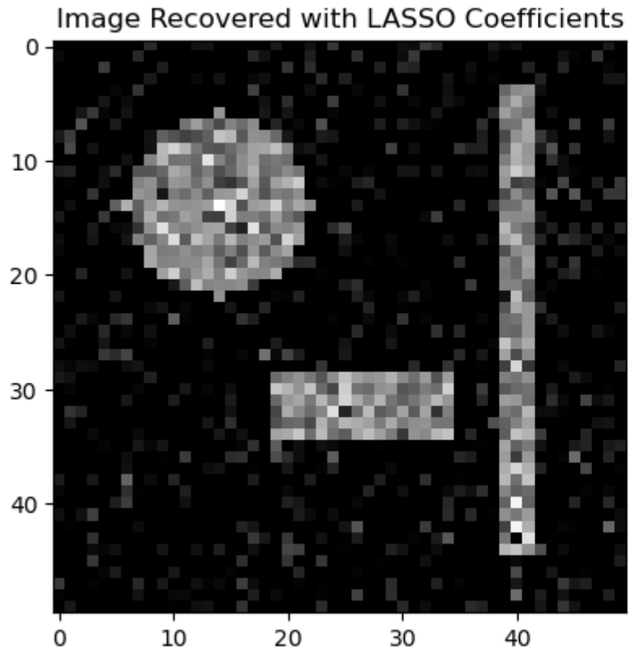


Image Recovered with LASSO Coefficients
0
10
20
30
40
0
10
20
30
40

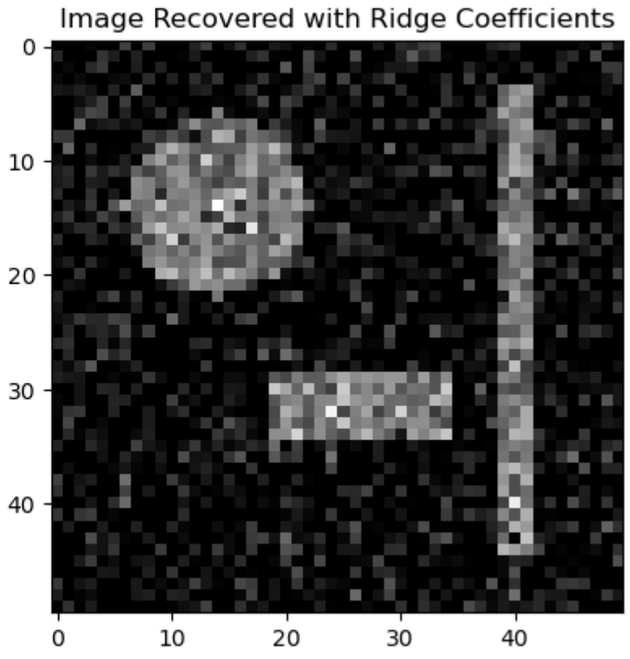


Image Recovered with Ridge Coefficients
0
10
20
30
40
0
10
20
30
40