

# COMP 6636 Mini-project 2

Dennis Brown, dgb0028@auburn.edu

April 23, 2021

## 1 Overview

This Mini-project looks at training accuracy and classification error using 2-layer Neural Network, Support Vector Machine, and Kernel Perceptron. The project was implemented in Python 3 and the code is included in section 6 of this report.

The project uses the Modified National Institute of Standards and Technology (MNIST) database of 60,000 handwritten digits 0 to 9, where the samples are 780-pixel bitmaps of the digit images. The data was provided in the “libsvm” format. Although there is a Python wrapper for libsvm that imports this data, that seems too heavyweight for this project, so instead a simple custom data importer reads the data into numpy arrays where each row is a sample and each column is a feature. For all runs, the data was split into 70% training and 30% testing data.

For each of the three classifiers:

- First, the classifier was trained and tested with many variations of hyperparameters in order to find the optimal (or near-optimal) values of each hyperparameter. In the interest of time, these trials were run on a small subset of the data (1000 samples split 70/30).
- Second, the classifier was trained and tested on the entire dataset (60000 samples split 70/30) using the best hyperparameters identified in the first step.

## 2 2-layer Neural Network

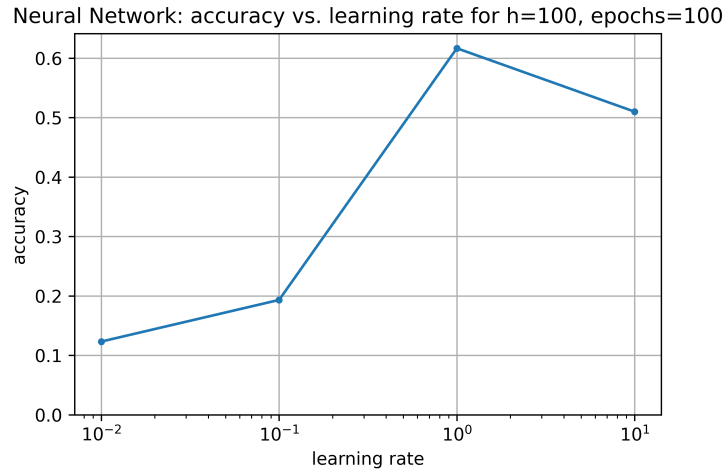
The 2-layer Neural Network uses a hidden layer of 100 units (as specified), although other numbers of hidden units were tested as well. The link/activation function is Sigmoid. A bias term was added to each input sample and output of the hidden layer. The decimal training labels were converted to 4-digit binary numbers and the network has 4 inputs and 4 outputs. The outputs are then converted back to decimal.

### 2.1 Finding optimal parameters

#### 2.1.1 Learning Rate

The learning rate was varied from 0.01 to 10.0 with fixed values of 100 hidden layers and 100 epochs. 1.0 performed best.

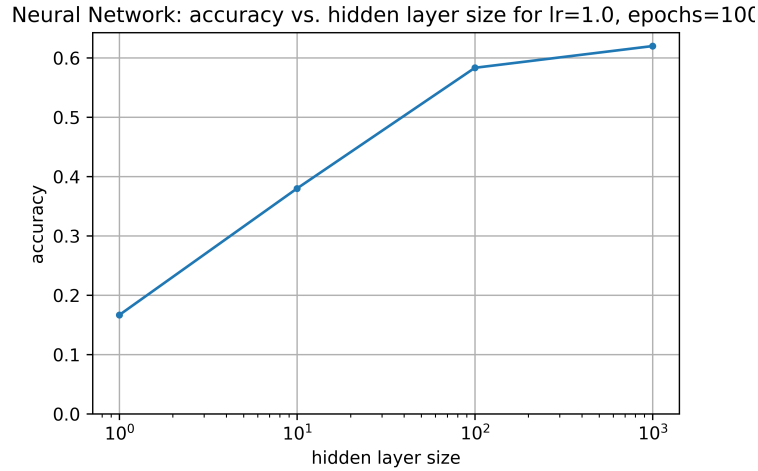
Figure 1: Neural Network: accuracy vs. learning rate



### 2.1.2 Hidden Units

The size of the hidden layer was varied from 1 to 1000 with fixed values of 1.0 learning rate and 100 epochs. 1000 units performed best, but only slightly better than 100 and with a much higher computational cost. 100 hidden units were used in all other Neural Network trials.

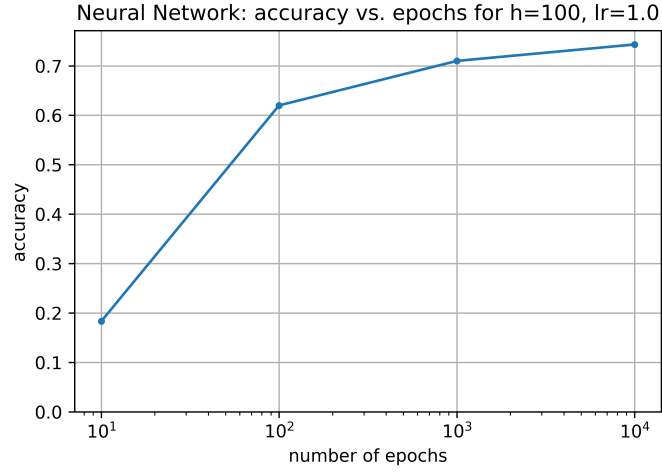
Figure 2: Neural Network: accuracy vs. hidden units



### 2.1.3 Epochs

The number of epochs was varied from 10 to 10000 with fixed values of 1.0 learning rate and 100 hidden units. 10000 epochs performed best, but only slightly better than 1000 and with a much higher computational cost. 1000 epochs were used in all other Neural Network trials.

Figure 3: Neural Network: accuracy vs. epochs



## 2.2 Performance on entire dataset

Using the entire dataset, the Neural Network was trained on 42000 samples and tested on 18000 samples using 100 hidden layers, 1.0 learning rate, and 1000 epochs.

The accuracy was 15489/18000 (86.05%).

The confusion matrix is shown below. One can observe “hotspots” where the digit 5 was misclassified as 1, 8 as 9, 8 as 0, and 3 as 1 and 2. The author isn’t quite sure why that would happen other than noting those digits can look similar when handwritten.

Figure 4: Confusion Matrix: NN on entire dataset

		Actual									
Predicted		0	1	2	3	4	5	6	7	8	9
	0	1663	19	66	15	41	32	18	4	99	23
	1	17	1879	4	100	16	137	1	24	63	71
	2	26	4	1579	93	1	11	42	13	25	2
	3	5	12	40	1503	1	30	0	68	20	19
	4	27	2	17	2	1473	72	55	14	22	15
	5	4	14	3	35	64	1211	4	83	9	55
	6	8	0	56	2	36	23	1609	18	2	0
	7	0	2	19	50	2	82	4	1680	10	22
	8	25	6	18	10	47	5	3	0	1440	80
	9	5	5	9	53	62	34	2	31	121	1452

## 3 Linear Support Vector Machine

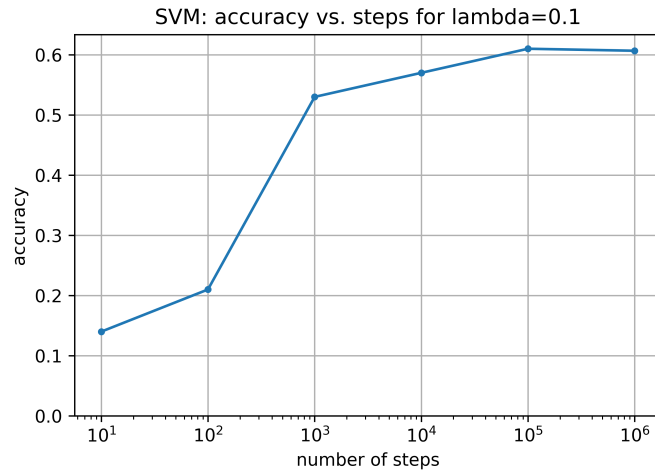
The linear SVM is a binary classifier. Employing a form of Error-Correcting Output Code (ECOC), the decimal training labels were converted to 4-digit binary numbers and fed into four instances of the linear SVM. The output of those 4 instances were then combined and converted back to decimal.

### 3.1 Finding optimal parameters

#### 3.1.1 Steps

The number of steps was varied from 10 to 1000000 with a fixed value of 0.1 lambda. 100000 steps performed best.

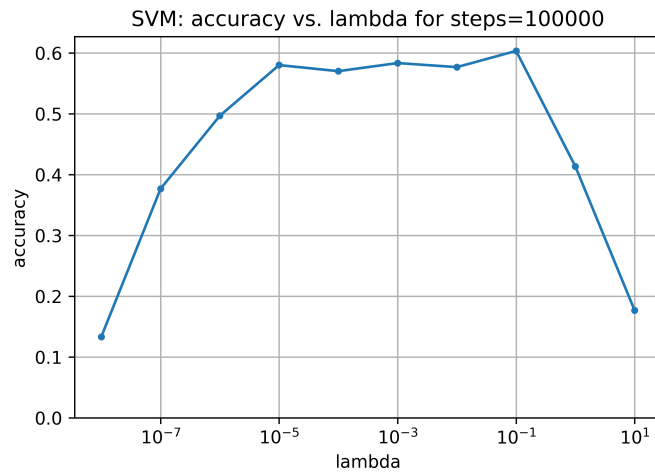
Figure 5: SVM: accuracy vs. steps



#### 3.1.2 Lambda

The value of lambda was varied from 1e-8 to 10 with a fixed value of 100,000 steps. Lambda = 0.1 performed best.

Figure 6: SVM: accuracy vs. lambda



### 3.2 Performance on entire dataset

Using the entire dataset, the four SVMs were trained on 42000 samples and tested on 18000 samples using 100000 steps and lambda = 0.1.

The accuracy was 11473/18000 (63.74%).

The confusion matrix is shown below. One should note a lot of confusion here, as would be indicated by the mediocre performance in accuracy. The worst offenses seem to be misclassifying 8 as 0, 5 as 1, 7 as 5, and 3 as 1.

Figure 7: Confusion Matrix: SVM on entire dataset

		Actual									
Predicted		0	1	2	3	4	5	6	7	8	9
	0	1402	43	152	44	39	165	88	11	461	120
	1	69	1834	30	376	48	445	4	110	337	186
	2	175	4	1384	121	6	40	94	21	52	20
	3	5	22	92	1155	1	75	1	246	36	28
	4	79	2	24	7	1144	176	122	23	51	51
	5	4	8	5	16	309	515	1	424	45	231
	6	36	0	71	18	62	51	1419	16	11	5
	7	0	4	20	49	5	63	4	1008	4	30
	8	8	25	23	13	9	31	4	5	695	151
	9	2	1	10	64	120	76	1	71	119	917

## 4 Kernel Perceptron

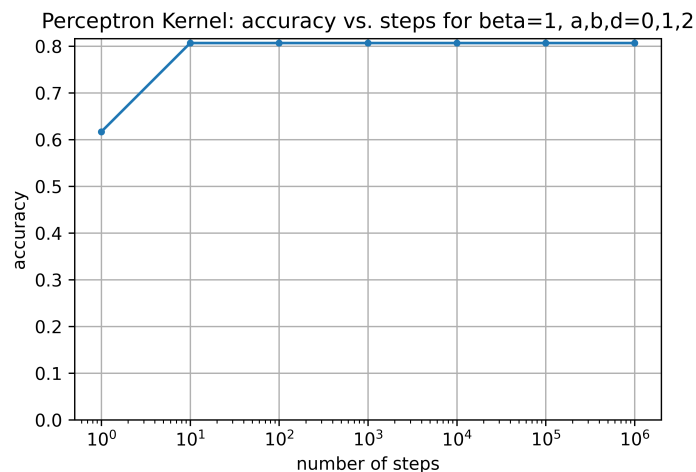
The Kernel Perceptron was implemented with a polynomial kernel  $K_{poly}^d(x, z) = (a + bx^T z)^d$ . It is a binary classifier. Employing a form of Error-Correcting Output Code (ECOC), the decimal training labels were converted to 4-digit binary numbers and fed into four instances of Kernel Perceptron. The output of those 4 instances were then combined and converted back to decimal.

### 4.1 Finding optimal parameters

#### 4.1.1 Steps

The number of steps was varied from 1 to 1000000 with fixed values of  $\beta = 1.0$  and the polynomial kernel variables  $a, b, d = 0.0, 1.0, 2.0$ . There was no improvement after 10 steps.

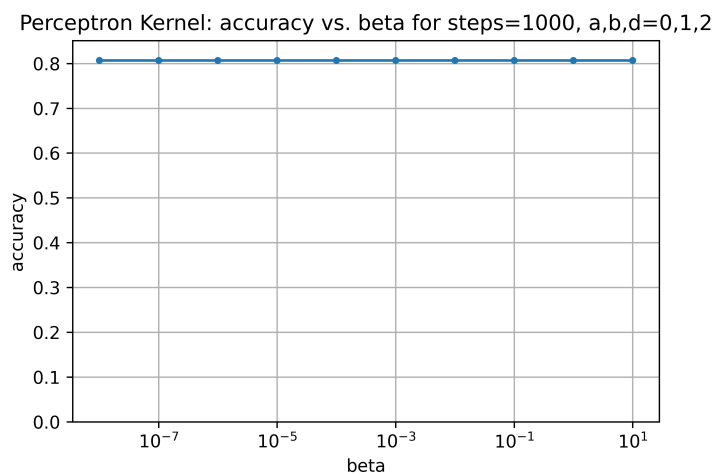
Figure 8: KP: accuracy vs. steps



### 4.1.2 Beta

The value of beta was varied from  $1e-8$  to 10 with fixed values of 1000 steps and the polynomial kernel variables  $a, b, d = 0.0, 1.0, 2.0$ . The value of beta does not appear to matter. I double-checked and my code does use it.

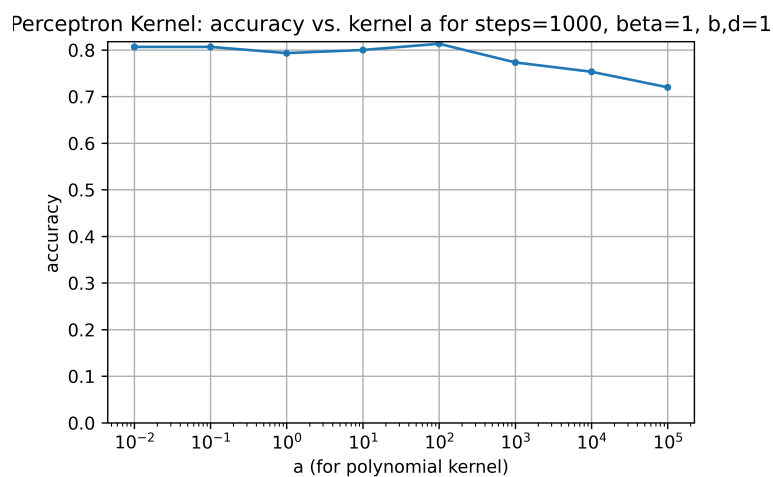
Figure 9: KP: accuracy vs beta



### 4.1.3 Kernel a

The value of  $a$  in the polynomial kernel was varied from 0.01 to 100000 with fixed values of 1000 steps,  $\beta = 1.0$ , and the polynomial kernel variables  $b, d = 1.0, 2.0$ .  $a = 100$  performed best.

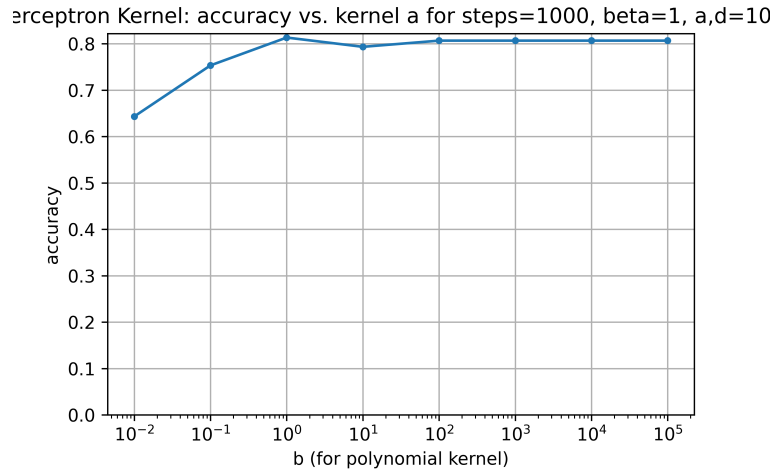
Figure 10: KP: accuracy vs kernel a



### 4.1.4 Kernel b

The value of  $b$  in the polynomial kernel was varied from 0.01 to 100000 with fixed values of 1000 steps,  $\beta = 1.0$ , and the polynomial kernel variables  $a, d = 100, 2.0$ .  $b = 1.0$  performed best.

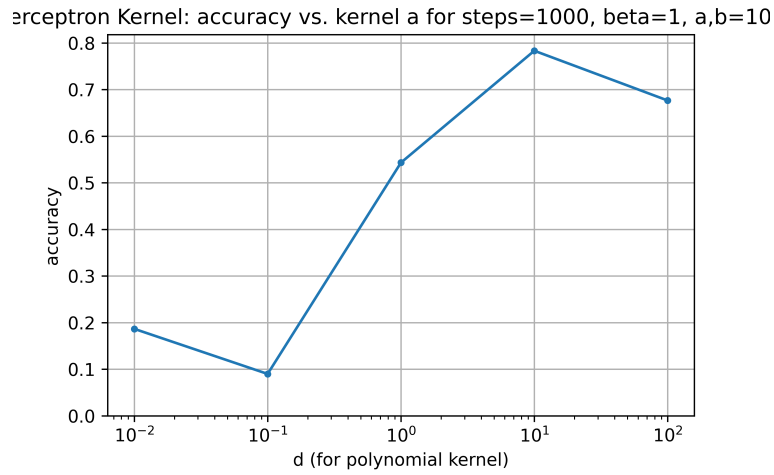
Figure 11: KP: accuracy vs kernel b



#### 4.1.5 Kernel d

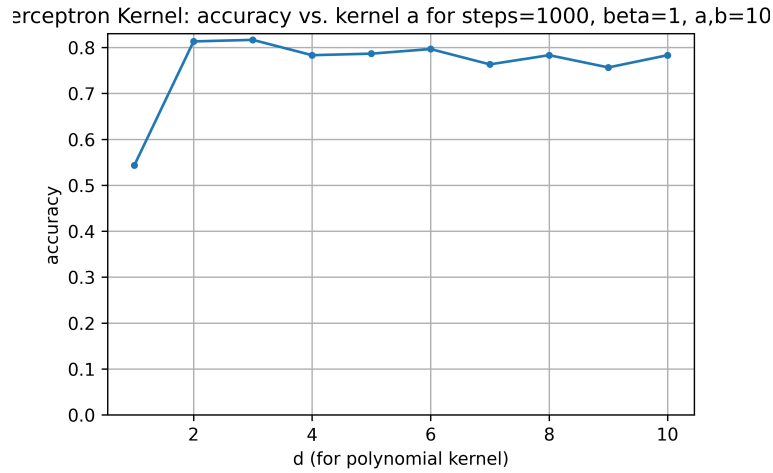
The value of d in the polynomial kernel was first varied from 0.01 to 100 with fixed values of 1000 steps,  $\beta = 1.0$ , and the polynomial kernel variables a, b = 100, 1.0. d = 10 performed best.

Figure 12: KP: accuracy vs kernel d



However, prior runs showed there was reason to further explore d in the range between 0 and 10. In the end, d = 3 performed best.

Figure 13: KP: accuracy vs kernel d part 2



## 4.2 Performance on entire dataset

Using the entire dataset, the four Kernel Perceptrons were trained on 42000 samples and tested on 18000 samples using 10 steps,  $\beta = 1.0$ , and polynomial kernel values  $a = 100$ ,  $b = 1.0$ , and  $d = 3.0$ .

The accuracy was 17013/18000 (94.52%).

The confusion matrix is shown below. The worst misclassifications are well under 100 each (much less than SVM) and include misclassifying 8 as 9, 5 as 7, 7 as 3, and 4 as 0.

Figure 14: Confusion Matrix: Kernel Perceptron on entire dataset

		Actual									
		0	1	2	3	4	5	6	7	8	9
Predicted	0	1740	9	22	1	47	8	4	1	43	10
	1	9	1912	1	27	7	40	0	18	11	30
	2	14	4	1742	33	3	3	23	5	18	0
	3	0	7	16	1748	0	11	0	54	5	9
	4	6	0	0	0	1618	18	9	5	2	6
	5	0	2	0	5	13	1461	0	18	2	12
	6	1	2	10	0	18	12	1693	16	0	0
	7	0	1	1	10	2	60	4	1801	0	7
	8	8	1	5	8	14	3	1	1	1667	34
	9	2	5	14	31	21	21	4	16	63	1631

## 5 Conclusion

Three classifiers were presented along with an investigation of their hyperparameters. The best-configured Kernel Perceptron had the best accuracy on the entire MNIST dataset of the three classifiers and took about 5 hours of computing time to train and test. The best-configured 2-layer Neural Network followed as a close second in accuracy, but took about 12 hours to train and test. The best-configured SVM had very mediocre accuracy, but only took about 1 hour to train and test.



## 6 Code

---

```
1 # -*- coding: utf-8 -*-
2 """
3 Mini project 2
4
5 Dennis Brown, COMP6636, 23 APR 2021
6 """
7
8 import numpy as np
9 import random
10 import sys
11 import matplotlib.pyplot as plt
12
13
14 #####
15 #
16 # MNIST data set functions
17 #
18 #####
19
20 def libsvm_scale_import(filename, limit = 0):
21     """
22     Read data from a libsvm .scale file. Set 'limit' to limit the import
23     to a certain number of samples.
24     """
25     datafile = open(filename, 'r')
26
27     # First pass: get dimensions of data
28     num_samples = 0
29     max_feature_id = 0
30     for line in datafile:
31         num_samples += 1
32         tokens = line.split()
33         for feature in tokens[1:]:
34             feature_id = int(feature.split(':')[0])
35             max_feature_id = max(feature_id, max_feature_id)
36
37     # Second pass: read data into array
38     # If the limit is set, import up to the limit, otherwise import all
39     import_samples = min(num_samples, limit) if limit else num_samples
40     features = np.zeros((import_samples, max_feature_id))
41     classes = np.zeros((import_samples, 1))
42     curr_sample = 0
43     # Read the samples
44     datafile.seek(0)
45     for line in datafile:
46         # Stop at the limit if it's set
47         if (limit and (curr_sample >= limit)): break
48         tokens = line.split()
49         # Read the classification
50         classes[curr_sample][0] = float(tokens[0])
51         # Read the features
52         for feature in tokens[1:]:
53             feature_id = int(feature.split(':')[0])
54             feature_val = float(feature.split(':')[1])
55             features[curr_sample][feature_id - 1] = feature_val
56         curr_sample += 1
57     datafile.close()
58
59     print('LOADED:', filename, ':', classes.shape, features.shape)
60
61     return classes, features
62
63
64 def convert_mnist_classes_to_binary(classes):
65     """
66     Given a list of integer MNIST classes, return an array where each class is
```

```

67     converted to binary. e.g., 5 -> [0. 1. 0. 1.]
68     """
69     binary_classes = np.zeros((classes.shape[0], 4))
70     for i in range(classes.shape[0]):
71         boolver = bin(int(classes[i][0]))[2:].zfill(4)
72         for bit in range(len(boolver)): binary_classes[i][bit] = float(boolver[bit])
73         # print(classes[i][0], binary_classes[i])
74     return binary_classes
75
76
77 def convert_mnist_classes_to_integer(binary_classes):
78     """
79     Given a list of binary MNIST classes, return an array where each class is
80     converted to integer. e.g., [0. 1. 0. 1.] -> 5
81     """
82     classes = np.zeros((binary_classes.shape[0], 1))
83     for i in range(binary_classes.shape[0]):
84         bins = binary_classes[i]
85         # Not very elegant, but it works — convert to integer and cap at 9
86         classes[i][0] = min((bins[0] * 8) + (bins[1] * 4) + (bins[2] * 2) + bins[3], 9)
87         # print(classes[i][0], binary_classes[i])
88     return classes
89
90
91 #####
92 #
93 # Neural Network functions
94 #
95 #####
96
97 def sigmoid(x):
98     """
99     Sigmoid Function
100     """
101     return 1 / (1 + np.exp(-x))
102
103
104 def train_neural_network(X, y, H_size, learning_rate, epochs):
105     """
106     2-Layer Neural Network: Given a 2-D set of data X (samples are rows,
107     columns features), a vector y of classifications, a number of
108     hidden-layer neurons, a learning rate, and number of epochs,
109     train a 2-layer neural network. Return weight matrices.
110     """
111     # Randomly initialize the weights for the input -> hidden layer
112     xh = (np.random.random((X.shape[1] + 1, H_size))) * 2 - 1
113
114     # Randomly initialize the weights for the hidden layer -> output
115     hy = (np.random.random((H_size + 1, y.shape[1]))) * 2 - 1
116
117     for epoch in range(epochs):
118         # print(str(epoch) + ' ', end = '')
119         # sys.stdout.flush()
120
121         # -----
122         # Forward Propagation
123         # -----
124
125         # Add bias terms to X
126         X_bias = np.hstack((X, np.ones([X.shape[0], 1])))
127
128         # Calculate hidden layer outputs
129         H_output = sigmoid(np.dot(X_bias, xh))
130
131         # Add bias terms to H_output
132         H_output_bias = np.hstack((H_output, (np.ones([H_output.shape[0], 1]))))
133
134         y_hat = sigmoid(np.dot(H_output_bias, hy))

```

```

135
136     # -----
137     # Backward Propagation
138     # -----
139
140     # Find error
141     y_error = y_hat - y
142
143     # Calculate hidden layer error (remove bias from hy)
144     H_error = H_output * (1 - H_output) * np.dot(y_error, hy.T[:, 1:])
145
146     # Calculate partial derivatives
147     H_pd = X_bias[:, :, np.newaxis] * H_error[:, np.newaxis, :]
148     y_pd = H_output.bias[:, :, np.newaxis] * y_error[:, np.newaxis, :]
149
150     # Calculate total gradients for hidden and output layers
151     # (find average of each column)
152     H_gradient = np.average(H_pd, axis = 0)
153     y_gradient = np.average(y_pd, axis = 0)
154
155     # Update weights using learning rate and gradients
156     xh -= (learning_rate * H_gradient)
157     hy -= (learning_rate * y_gradient)
158
159     # print()
160
161     # Return weight matrices when finished
162     return xh, hy
163
164
165 def test_neural_network(X, xh, hy):
166     """
167     2-layer Neural Network: Given a 2-D set of data X (samples are rows,
168     columns features) and weight matrices for the 2-layer network,
169     return the predicted output.
170     """
171     X_bias = np.hstack((X, np.ones([X.shape[0], 1])))
172     H_output = sigmoid(np.dot(X_bias, xh))
173     H_output.bias = np.hstack((H_output, (np.ones([H_output.shape[0], 1]))))
174     y_hat = sigmoid(np.dot(H_output.bias, hy))
175     return y_hat
176
177
178 def mnist_neural_network(train_classes, test_classes, train_features,
179                           test_features, h_size, learning_rate, num_epochs):
180     """
181     Given MNIST features and classes split into training and testing data,
182     train and evaluate Neural Network.
183     """
184     # Convert classifications to binary
185     binary_train_classes = convert_mnist_classes_to_binary(train_classes)
186
187     # Train
188     xh, hy = train_neural_network(train_features, binary_train_classes,
189                                   h_size, learning_rate, num_epochs)
190
191     # Test
192     binary_pred_classes = test_neural_network(test_features, xh, hy)
193     binary_pred_classes = 1.0 * (binary_pred_classes > 0.5)
194     pred_classes = convert_mnist_classes_to_integer(binary_pred_classes)
195
196     # Create label for this evaluation
197     label = str(train_features.shape[0] + test_features.shape[0])
198     label += '_' + str(h_size)
199     label += '_' + str(learning_rate)
200     label += '_' + str(num_epochs)
201
202     # Calculate number correct

```

```

203     correct = 0
204     cm = np.zeros((10, 10))
205     for i in range(test_classes.shape[0]):
206         if (pred_classes[i][0] == test_classes[i][0]): correct += 1
207         cm[int(pred_classes[i][0])][int(test_classes[i][0])] += 1
208     print('Correct:', correct, '/', test_classes.shape[0], 'for', label)
209     print(cm)
210
211     np.savetxt('./data/confusion_nn_' + label + '.csv', cm, delimiter=',', fmt='%10.0f')
212
213     return correct / test_classes.shape[0]
214
215 #####
216 #
217 # Support Vector Machine functions
218 #
219 #####
220
221 def train_svm(X, y, lam, limit):
222     """
223     Support Vector Machine. Given a sample matrix X,
224     a vector Y of classifications, a regularization parameter lam,
225     and a step limit, train and return a weight vector that
226     can be used to classify the given data.
227     """
228     # Convert (1, 0) to (1, -1)
229     y = y * 2 - 1
230
231     # Initialize the weight vector
232     w = np.zeros(X.shape[1])
233
234     # Pegasos algorithm
235     # Repeat the main loop until we reach the iteration limit
236     t = 1
237     while(t <= limit):
238         i = random.randint(0, X.shape[0] - 1)
239         eta = 1.0 / (lam * t)
240         y_hat = y[i][0] * np.matmul(w, X[i])
241         if (y_hat < 1.0):
242             w = ((1 - (eta * lam)) * w) + (eta * y[i][0] * X[i])
243         else:
244             w = ((1 - (eta * lam)) * w)
245         if (np.linalg.norm(w) > 0.0):
246             w = min(1.0, ((1.0 / np.sqrt(lam)) / (np.linalg.norm(w)))) * w
247         t += 1
248
249     return w
250
251 def test_svm(X, w):
252     """
253     Support Vector Machine. Given a sample matrix X
254     and a weight vector, predict the classes of X.
255     """
256     # Calculate predictions
257     y_hat = np.zeros((X.shape[0], 1))
258     for i in range(X.shape[0]):
259         y_hat[i][0] = np.matmul(w, X[i])
260
261     # Convert to (1, -1)
262     y_hat = np.sign(y_hat)
263
264     # Convert (1, -1) to (1, 0)
265     y_hat = (y_hat + 1) / 2
266
267     return y_hat
268
269
270

```

```

271
272 def mnist_svm(train_classes, test_classes, train_features, test_features,
273               limit, lam):
274     """
275     Given MNIST features and classes split into training and testing data,
276     train and evaluate Support Vector Machine.
277     """
278     # Convert classes to four binary y vectors
279     binary_train_classes = convert_mnist_classes_to_binary(train_classes)
280     y1 = binary_train_classes[:, [0]]
281     y2 = binary_train_classes[:, [1]]
282     y3 = binary_train_classes[:, [2]]
283     y4 = binary_train_classes[:, [3]]
284
285     # Train on the four y vectors
286     w1 = train_svm(train_features, y1, lam, limit)
287     w2 = train_svm(train_features, y2, lam, limit)
288     w3 = train_svm(train_features, y3, lam, limit)
289     w4 = train_svm(train_features, y4, lam, limit)
290
291     # Get binary predictions from the four perceptrons
292     y_hat1 = test_svm(test_features, w1)
293     y_hat2 = test_svm(test_features, w2)
294     y_hat3 = test_svm(test_features, w3)
295     y_hat4 = test_svm(test_features, w4)
296
297     # Convert binary predictions back to decimal
298     binary_pred_classes = np.hstack((y_hat1, y_hat2, y_hat3, y_hat4))
299     pred_classes = convert_mnist_classes_to_integer(binary_pred_classes)
300
301     # Create label for this evaluation
302     label = str(train_features.shape[0] + test_features.shape[0])
303     label += '-' + str(limit)
304     label += '-' + str(lam)
305
306     # Calculate number correct
307     correct = 0
308     cm = np.zeros((10, 10))
309     for i in range(test_classes.shape[0]):
310         if (pred_classes[i][0] == test_classes[i][0]): correct += 1
311         cm[int(pred_classes[i][0])][int(test_classes[i][0])] += 1
312     print('Correct:', correct, '/', test_classes.shape[0], 'for', label)
313     print(cm)
314
315     np.savetxt('./data/confusion_svm-' + label + '.csv', cm, delimiter=',', fmt='%10.0f')
316
317     return correct / test_classes.shape[0]
318
319 #####
320 #
321 # Perceptron Kernel functions
322 #
323 #####
324 #####
325
326 def poly_kernel(x, z, a, b, d):
327     """
328     Calculate polynomial kernel for samples x and z.
329     a, b, and d are hyperparameters.
330     """
331     return (a + (b * (np.matmul(x.T, z))) ** d)
332
333
334 def gram(X, ka, kb, kd):
335     """
336     Calculate Gram Matrix given X and parameters for poly kernel
337     """
338

```

```

339 G = np.zeros((X.shape[0], X.shape[0]))
340 for i in range(X.shape[0]):
341     for j in range(X.shape[0]):
342         G[i][j] = poly_kernel(X[i], X[j], ka, kb, kd)
343
344 return G
345
346
347 def train_perceptron_kernel(G, y, beta, step_limit):
348     """
349     Perceptron with a kernel. Given a Gram matrix G,
350     a vector Y of classifications, a learning rate (beta),
351     and a step limit, train and return a weight vector that
352     can be used to classify the given data.
353     """
354     # Convert (1, 0) to (1, -1)
355     y = y * 2 - 1
356
357     # Initialize the alpha vector
358     a = np.zeros(G.shape[0])
359
360     # Initialize y-hat
361     y_hat = np.zeros((G.shape[0], 1))
362
363     # Repeat the main loop until we have convergence or reach the
364     # iteration limit
365     steps = 0
366     converged = False
367     while(not(converged) and (steps < step_limit)):
368         converged = True # assume converged until we determine otherwise
369
370         # For each sample in X, calculate alpha's classification error
371         # and update alpha.
372         for i in range(G.shape[0]):
373
374             # Find current prediction based on kernel
375             y_hat[i][0] = np.sign(np.matmul(G[i, :], a))
376
377             # If error on this element is > a very small value (is not
378             # effectively 0), we need to update alpha, and have not converged.
379             error = y[i][0] - y_hat[i][0]
380             if (abs(error) > 0.000001):
381                 a[i] += beta * y[i][0]
382                 converged = False
383         steps += 1
384
385     return a
386
387
388 def test_perceptron_kernel(Xtrain, Xtest, a, ka, kb, kd):
389     """
390     Perceptron with a kernel. Given a sample matrices Xtrain and Xtest,
391     and vector a, return predicted classes.
392     """
393     y_hat = np.zeros((Xtest.shape[0], 1))
394
395     for i in range(Xtest.shape[0]):
396         for j in range(a.shape[0]):
397             y_hat[i][0] += a[j] * poly_kernel(Xtrain[j], Xtest[i], ka, kb, kd)
398
399     # Convert to (1, -1)
400     y_hat = np.sign(y_hat)
401
402     # Convert (1, -1) to (1, 0)
403     y_hat = (y_hat + 1) / 2
404
405     return y_hat
406

```

```

407
408 def mnist_perceptron_kernel(train_classes, test_classes, train_features,
409                             test_features, limit, beta, ka, kb, kd):
410     """
411     Given MNIST features and classes split into training and testing data,
412     train and evaluate Kernel Perceptron. ka, kb, and kd are for poly kernel.
413     """
414     # Convert classes to four binary y vectors
415     binary_train_classes = convert_mnist_classes_to_binary(train_classes)
416     y1 = binary_train_classes[:, [0]]
417     y2 = binary_train_classes[:, [1]]
418     y3 = binary_train_classes[:, [2]]
419     y4 = binary_train_classes[:, [3]]
420
421     # Train on the four y vectors
422     G = gram(train_features, ka, kb, kd)
423     a1 = train_perceptron_kernel(G, y1, beta, limit)
424     a2 = train_perceptron_kernel(G, y2, beta, limit)
425     a3 = train_perceptron_kernel(G, y3, beta, limit)
426     a4 = train_perceptron_kernel(G, y4, beta, limit)
427
428     # Get binary predictions from the four perceptrons
429     y_hat1 = test_perceptron_kernel(train_features, test_features, a1, ka, kb, kd)
430     y_hat2 = test_perceptron_kernel(train_features, test_features, a2, ka, kb, kd)
431     y_hat3 = test_perceptron_kernel(train_features, test_features, a3, ka, kb, kd)
432     y_hat4 = test_perceptron_kernel(train_features, test_features, a4, ka, kb, kd)
433
434     # Convert binary predictions back to decimal
435     binary_pred_classes = np.hstack((y_hat1, y_hat2, y_hat3, y_hat4))
436     pred_classes = convert_mnist_classes_to_integer(binary_pred_classes)
437
438     # Create label for this evaluation
439     label = str(train_features.shape[0] + test_features.shape[0])
440     label += '_' + str(limit)
441     label += '_' + str(beta)
442     label += '_' + str(ka)
443     label += '_' + str(kb)
444     label += '_' + str(kd)
445
446     # Calculate number correct
447     correct = 0
448     cm = np.zeros((10, 10))
449     for i in range(test_classes.shape[0]):
450         if (pred_classes[i][0] == test_classes[i][0]): correct += 1
451         cm[int(pred_classes[i][0])][int(test_classes[i][0])] += 1
452     print('Correct:', correct, '/', test_classes.shape[0], 'for', label)
453     print(cm)
454
455     np.savetxt('./data/confusion_kp_' + label + '.csv', cm, delimiter=',', fmt='%10.0f')
456
457     return correct / test_classes.shape[0]
458
459 #####
460 #
461 # Run it all
462 #
463 #####
464
465 def main():
466     #####
467     #
468     # PART 1: VARIATIONS OF HYPERPARAMETERS ON A SMALL DATA SET
469     #
470     #####
471
472     # Load small data set for variation tests
473
474

```

```

475 sample_limit = 1000
476 classes, features = libsvm_scale_import('data/mnist.scale', limit = sample_limit)
477 split = int(len(classes) * 0.70)
478 train_classes = classes[:split]
479 test_classes = classes[split:]
480 train_features = features[:split]
481 test_features = features[split:]
482 print('training data =', train_features.shape, train_classes.shape)
483 print('test_data =', test_features.shape, test_classes.shape)
484
485 # Test decimal-binary-decimal conversion
486 binary_train_classes = convert_mnist_classes_to_binary(train_classes)
487 decimal_train_classes = convert_mnist_classes_to_integer(binary_train_classes)
488 print(train_classes - decimal_train_classes)
489
490 # Execute Neural Network testing
491 print('\nNeural Network Variations')
492
493 # Vary learning rate
494 nn_lrs = np.array([0.01, 0.1, 1.0, 10.0])
495 nn_lr_results = np.zeros(nn_lrs.shape)
496 for i in range(nn_lrs.shape[0]):
497     nn_lr_results[i] = mnist_neural_network(train_classes, test_classes, train_features,
498                                           test_features, 100, nn_lrs[i], 100)
499
500 plt.clf()
501 plt.plot(nn_lrs, nn_lr_results, marker='.')
502 plt.title('Neural Network: accuracy vs. learning rate for h=100, epochs=100')
503 plt.xscale('log')
504 plt.xlabel('learning rate')
505 plt.ylabel('accuracy')
506 plt.ylim(bottom = 0)
507 plt.grid(True)
508 plt.savefig('./plots/nn-accuracy-learning-rate.png', dpi = 600)
509
510 # Vary size of hidden layer
511 nn_hs = np.array([1, 10, 100, 1000])
512 nn_h_results = np.zeros(nn_hs.shape)
513 for i in range(nn_hs.shape[0]):
514     nn_h_results[i] = mnist_neural_network(train_classes, test_classes, train_features,
515                                           test_features, nn_hs[i], 1.0, 100)
516
517 plt.clf()
518 plt.plot(nn_hs, nn_h_results, marker='.')
519 plt.title('Neural Network: accuracy vs. hidden layer size for lr=1.0, epochs=100')
520 plt.xscale('log')
521 plt.xlabel('hidden layer size')
522 plt.ylabel('accuracy')
523 plt.ylim(bottom = 0)
524 plt.grid(True)
525 plt.savefig('./plots/nn-accuracy_hsize.png', dpi = 600)
526
527 # Vary number of epochs
528 nn_epochs = np.array([10, 100, 1000, 10000])
529 nn_epoch_results = np.zeros(nn_epochs.shape)
530 for i in range(nn_epochs.shape[0]):
531     nn_epoch_results[i] = mnist_neural_network(train_classes, test_classes,
532                                           train_features,
533                                           test_features, 100, 1.0, nn_epochs[i])
534
535 plt.clf()
536 plt.plot(nn_epochs, nn_epoch_results, marker='.')
537 plt.title('Neural Network: accuracy vs. epochs for h=100, lr=1.0')
538 plt.xscale('log')
539 plt.xlabel('number of epochs')
540 plt.ylabel('accuracy')
541 plt.ylim(bottom = 0)
542 plt.grid(True)
543 plt.savefig('./plots/nn-accuracy-epochs.png', dpi = 600)
544
545 # Execute SVM testing

```



```

542 print('\nSupport Vector Machine Variations')
543
544 # Vary number of steps
545 svm_steps = np.array([10, 100, 1000, 10000, 100000, 1000000])
546 svm_step_results = np.zeros(svm_steps.shape)
547 for i in range(svm_steps.shape[0]):
548     svm_step_results[i] = mnist_svm(train_classes, test_classes, train_features,
549                                     test_features, svm_steps[i], 0.1)
550
551 plt.clf()
552 plt.plot(svm_steps, svm_step_results, marker='.')
553 plt.title('SVM: accuracy vs. steps for lambda=0.1')
554 plt.xscale('log')
555 plt.xlabel('number of steps')
556 plt.ylabel('accuracy')
557 plt.ylim(bottom = 0)
558 plt.grid(True)
559 plt.savefig('./plots/svm-accuracy-step.png', dpi = 600)
560
561 # Vary lambda
562 svm_lams = np.array([1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1.0, 10.0])
563 svm_lam_results = np.zeros(svm_lams.shape)
564 for i in range(svm_lams.shape[0]):
565     svm_lam_results[i] = mnist_svm(train_classes, test_classes, train_features,
566                                     test_features, 100000, svm_lams[i])
567
568 plt.clf()
569 plt.plot(svm_lams, svm_lam_results, marker='.')
570 plt.title('SVM: accuracy vs. lambda for steps=100000')
571 plt.xscale('log')
572 plt.xlabel('lambda')
573 plt.ylabel('accuracy')
574 plt.ylim(bottom = 0)
575 plt.grid(True)
576 plt.savefig('./plots/svm-accuracy-lambda.png', dpi = 600)
577
578 # Execute Kernel Perceptron testing
579 print('\nKernel Perceptron Variations')
580
581 # Vary number of steps
582 kp_steps = np.array([1, 10, 100, 1000, 10000, 100000, 1000000])
583 kp_step_results = np.zeros(kp_steps.shape)
584 for i in range(kp_steps.shape[0]):
585     kp_step_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
586                                                  train_features, test_features,
587                                                  kp_steps[i], 1,
588                                                  0.0, 1.0, 2.0)
589
590 plt.clf()
591 plt.plot(kp_steps, kp_step_results, marker='.')
592 plt.title('Perceptron Kernel: accuracy vs. steps for beta=1, a,b,d=0,1,2')
593 plt.xscale('log')
594 plt.xlabel('number of steps')
595 plt.ylabel('accuracy')
596 plt.ylim(bottom = 0)
597 plt.grid(True)
598 plt.savefig('./plots/kp-accuracy-step.png', dpi = 600)
599
600 # Vary beta
601 kp_betas = np.array([1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1.0, 10.0])
602 kp_beta_results = np.zeros(kp_betas.shape)
603 for i in range(kp_betas.shape[0]):
604     kp_beta_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
605                                                  train_features, test_features,
606                                                  1000, kp_betas[i],
607                                                  0.0, 1.0, 2.0)
608
609 plt.clf()
610 plt.plot(kp_betas, kp_beta_results, marker='.')
611 plt.title('Perceptron Kernel: accuracy vs. beta for steps=1000, a,b,d=0,1,2')
612 plt.xscale('log')
613 plt.xlabel('beta')

```

```

610 plt.ylabel('accuracy')
611 plt.ylim(bottom = 0)
612 plt.grid(True)
613 plt.savefig('./plots/kp-accuracy-beta.png', dpi = 600)
614
615 # Vary a
616 kp_kas = np.array([1e-2, 1e-1, 1.0, 10.0, 100, 1000, 10000, 100000])
617 kp_ka_results = np.zeros(kp_kas.shape)
618 for i in range(kp_kas.shape[0]):
619     kp_ka_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
620                                                 train_features, test_features,
621                                                 1000, 1.0,
622                                                 kp_kas[i], 1.0, 2.0)
623
624 plt.clf()
625 plt.plot(kp_kas, kp_ka_results, marker='.')
626 plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, b,d=1,2')
627 plt.xscale('log')
628 plt.xlabel('a (for polynomial kernel)')
629 plt.ylabel('accuracy')
630 plt.ylim(bottom = 0)
631 plt.grid(True)
632 plt.savefig('./plots/kp-accuracy-ka.png', dpi = 600)
633
634 # Vary b
635 kp_kbs = np.array([1e-2, 1e-1, 1.0, 10.0, 100, 1000, 10000, 100000])
636 kp_kb_results = np.zeros(kp_kbs.shape)
637 for i in range(kp_kbs.shape[0]):
638     kp_kb_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
639                                                 train_features, test_features,
640                                                 1000, 1.0,
641                                                 100, kp_kbs[i], 2.0)
642
643 plt.clf()
644 plt.plot(kp_kbs, kp_kb_results, marker='.')
645 plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, a,d=100,2')
646 plt.xscale('log')
647 plt.xlabel('b (for polynomial kernel)')
648 plt.ylabel('accuracy')
649 plt.ylim(bottom = 0)
650 plt.grid(True)
651 plt.savefig('./plots/kp-accuracy-kb.png', dpi = 600)
652
653 # Vary d
654 kp_kds = np.array([1e-2, 1e-1, 1.0, 10.0, 100])
655 kp_kd_results = np.zeros(kp_kds.shape)
656 for i in range(kp_kds.shape[0]):
657     kp_kd_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
658                                                 train_features, test_features,
659                                                 1000, 1.0,
660                                                 100, 1.0, kp_kds[i])
661
662 plt.clf()
663 plt.plot(kp_kds, kp_kd_results, marker='.')
664 plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, a,b=100,1')
665 plt.xscale('log')
666 plt.xlabel('d (for polynomial kernel)')
667 plt.ylabel('accuracy')
668 plt.ylim(bottom = 0)
669 plt.grid(True)
670 plt.savefig('./plots/kp-accuracy-kd.png', dpi = 600)
671
672 # Vary d again (different and linear range)
673 kp_kds = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
674 kp_kd_results = np.zeros(kp_kds.shape)
675 for i in range(kp_kds.shape[0]):
676     kp_kd_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
677                                                 train_features, test_features,
678                                                 1000, 1.0,
679                                                 100, 1.0, kp_kds[i])
680
681 plt.clf()

```

```

678 plt.plot(kp_kds, kp_kd_results, marker='.')
679 plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, a,b=100,1')
680 plt.xlabel('d (for polynomial kernel)')
681 plt.ylabel('accuracy')
682 plt.ylim(bottom = 0)
683 plt.grid(True)
684 plt.savefig('./plots/kp-accuracy-kd-part2.png', dpi = 600)
685
686 #####
687 #
688 # PART 2: TEST ENTIRE DATA SET ON OPTIMAL-ISH PARAMETERS
689 #
690 #####
691
692 # Load all data
693 classes, features = libsvm_scale_import('data/mnist.scale')
694 split = int(len(classes) * 0.70)
695 train_classes = classes[:split]
696 test_classes = classes[split:]
697 train_features = features[:split]
698 test_features = features[split:]
699 print('training data =', train_features.shape, train_classes.shape)
700 print('test data =', test_features.shape, test_classes.shape)
701
702 mnist_neural_network(train_classes, test_classes, train_features,
703                      test_features, 100, 1.0, 1000) # this takes 12 hours
704
705 mnist_svm(train_classes, test_classes, train_features, test_features, 6000000, 0.1)
706
707 mnist_perceptron_kernel(train_classes, test_classes,
708                         train_features, test_features,
709                         10, 1.0, 100, 1.0, 3.0)
710
711
712 if __name__ == '__main__':
713     main()

```

---

MiniProj2.py