

COMP 6636 Mini-project 1

Dennis Brown, dgb0028@auburn.edu

March 3, 2021

1 Overview

This Mini-project looks at classification error using k-Nearest Neighbors (kNN) and Perceptron, plus attribute importance using Perceptron. The project was implemented in Python 3 and the code is included in section 6 of this report. The output log from running the code is in section 7.

The project considers two data sets:

- a4a, obtained from <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#a4a>
- iris, obtained from <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#iris>

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.

The a4a data included an “a4a” data set used for training and an “a4a.t” data set used for testing. The “a4a.t” data set had one more feature than the “a4a” data set, so the “a4a” data set was augmented by an extra column of zeros (which is implied by the libsvm data file format).

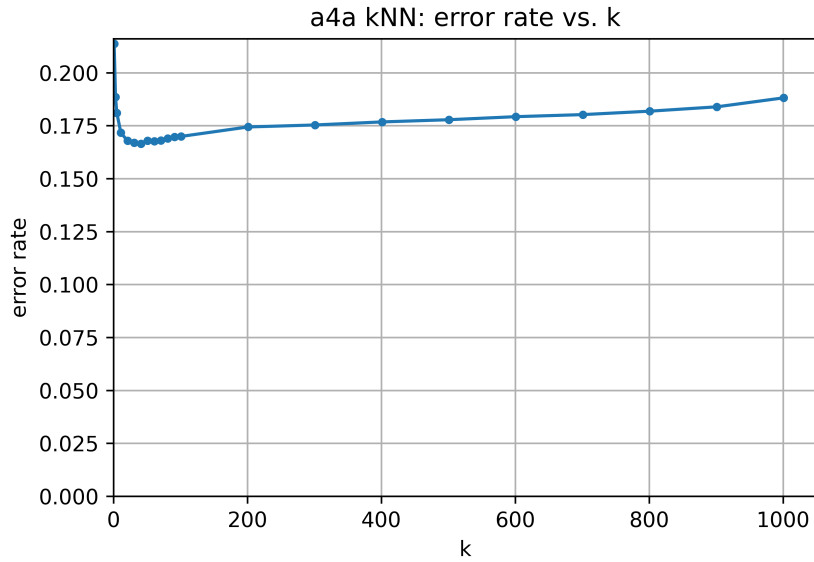
The iris data set didn’t include separate train and test data sets. It consists of 150 samples that are assigned to one of three classes and is sorted by class. Therefore, the data was randomly shuffled and partitioned into 50 training and 100 test samples (the shuffling ensures the three classes are represented in both train and test data sets).

2 Classification Error for a4a

2.1 kNN

a4a data was classified using kNN as described in the overview. There are 4781 samples in the training set and 27780 samples in the test set. The test data was classified for 22 values of k between 1 and 1001. No data reduction was performed and this testing took a few hours. The classification error rate improves until $k = 41$ (error rate 0.167), then it rises consistently for the remaining values of k .

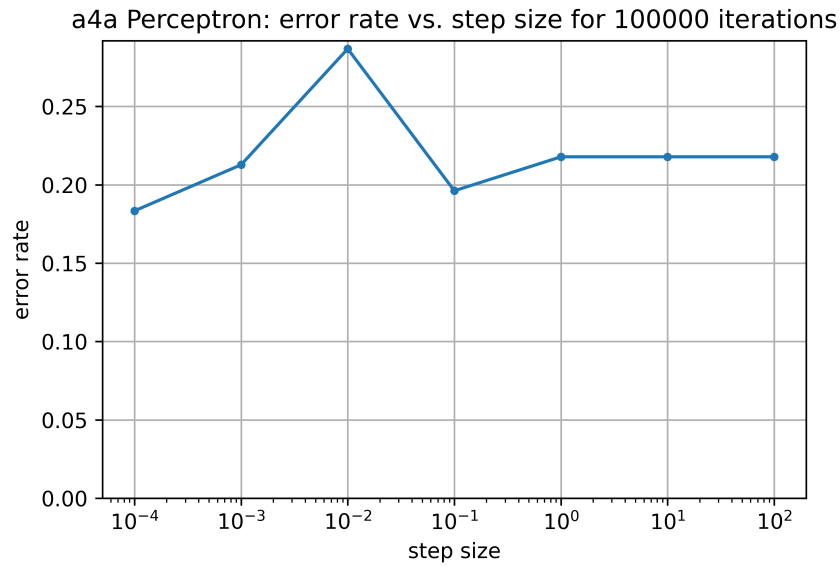
Figure 1: kNN classification error for a4a



2.2 Perceptron

a4a data was classified using Perceptron as described in the overview. There are 4781 samples in the training set and 27780 samples in the test set. The test data was classified for 7 values of β (step size) between .0001 and 100. The algorithm never converged with a limit of 100000 steps. No data reduction was performed. The lowest classification error rate is 0.183 with step size = 0.0001.

Figure 2: Perceptron classification error for a4a

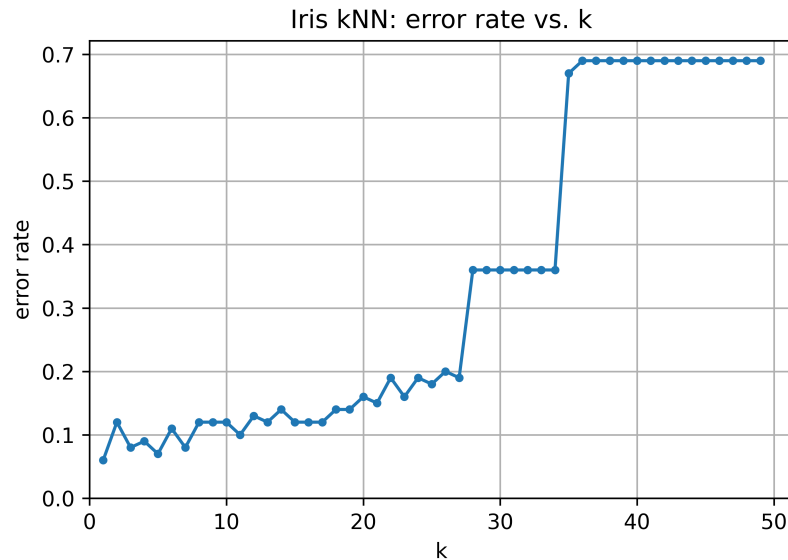


3 Classification Error for iris

3.1 kNN

iris data was classified using kNN as described in the overview. There are 50 samples in the training set and 100 samples in the test set. The minimum error is 0.06 at $k = 1$ and rises somewhat consistently from there. I don't have a good explanation for the stair-step effect starting at $k = 28$ and $k = 36$, other than it's an anomaly of how a small data set was randomly selected.

Figure 3: kNN classification error for iris



3.2 Perceptron

Since *iris* data has 3 classes, the One vs. All (OvA) approach was used to classify the data. Each of the three classes was trained and tested individually, where in each case the class being tested was considered “1” and the other classes were considered “-1”. Changing the step size β or the step limit didn't make any difference, so only one configuration is presented here with $\beta = 0.1$ and step limit = 100000. Multi-class perceptron provided 3 weight vectors as visualized in section 5.

Testing the three weight vectors give the following results:

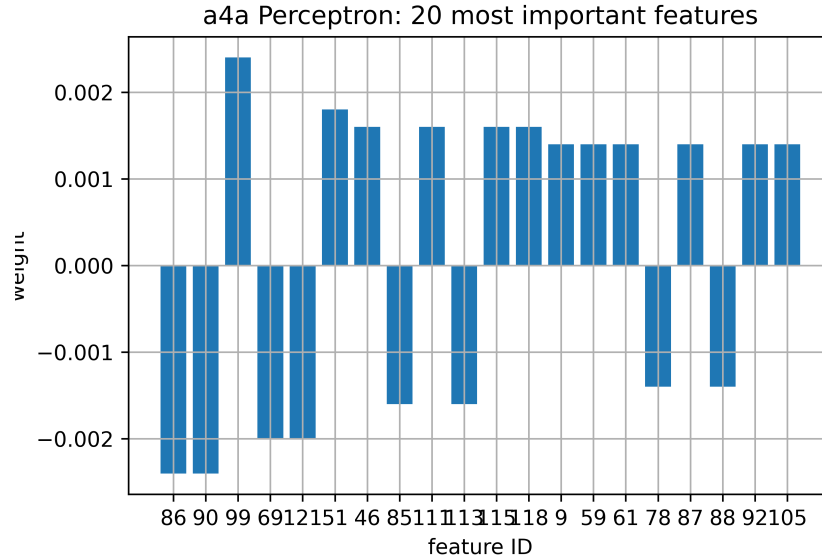
- Class 1: 0.00 error rate (it converged; class 1 is linearly separable from classes 2 and 3)
- Class 2: 0.68 error rate (did not converge)
- Class 3: 0.58 error rate (did not converge)
- Average: 0.42 error rate

I didn't graph the error rates because there are only three data points.

4 Importance of Attributes for a4a

The graph shows the top 20 feature ids after taking the weight vector associated with the lowest error rate and sorting the feature weights by magnitude. I could not find an explanation of what each feature id means. Feature 86 is the most important, whatever that is. I chose to visualize the actual values rather than absolute values so the reader can see what features are positively and negatively weighted.

Figure 4: a4a: Most important feature weights



5 Importance of Attributes for iris

Again, multi-class Perceptron was used to find a weight vector for each of the three classes of *iris* data. Since there are only four features, they are not shown here sorted for importance. It is more informative to look at the bar graph of each unsorted weight vector and see how the relative importance of each feature changes for the three classes:

- the signs of features 2 and 3 are reverse for classes 1 and 2;
- the sign of feature 4 is reverse between classes 2 and 3;
- the signs of features 2, 3, and 4 are reverse between classes 1 and 3; and
- feature 1 is relatively unimportant universally.

Figure 5: Feature weights for class = 1

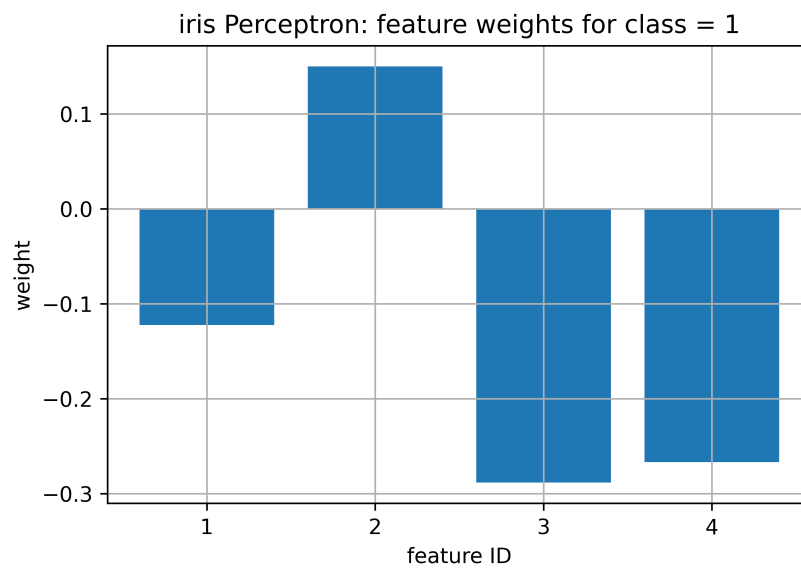


Figure 6: Feature weights for class = 2

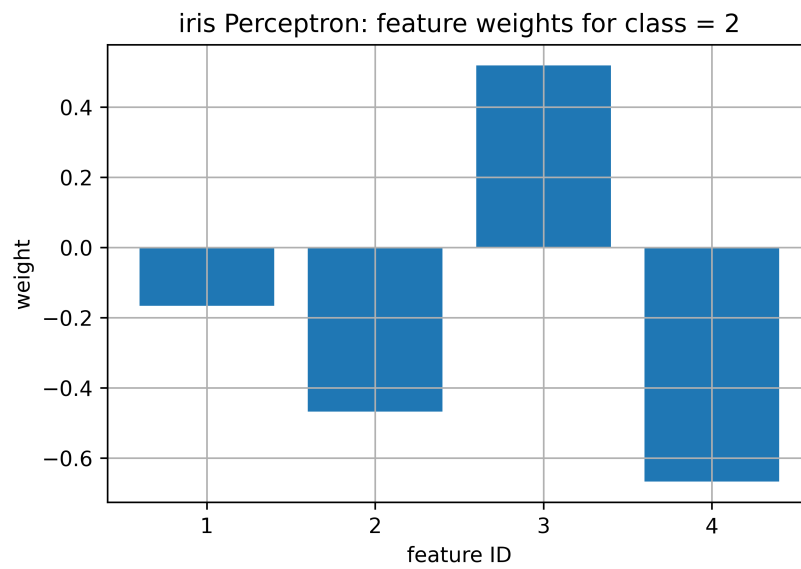
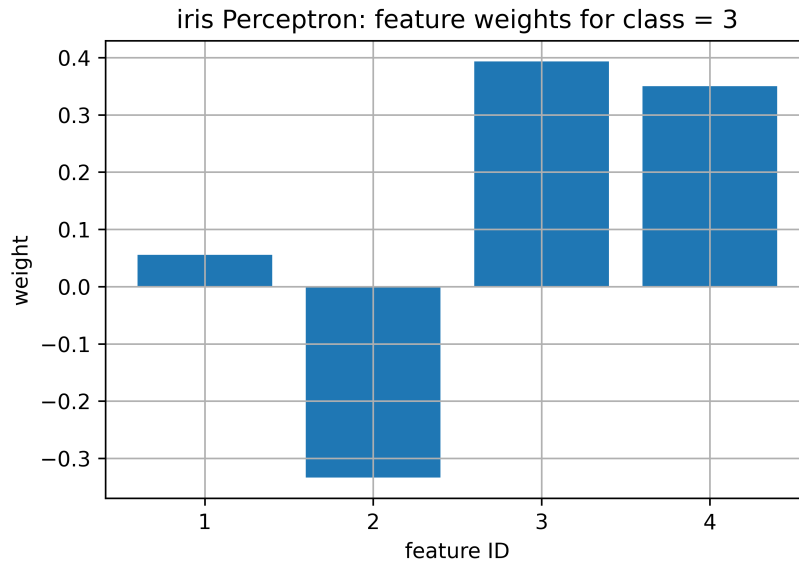


Figure 7: Feature weights for class = 3



6 Code

```

1  #-*- coding: utf-8 -*-
2  """
3  Mini project 1
4
5  Dennis Brown, COMP6636, 03 MAR 2021
6  """
7
8  import numpy as np
9  import copy
10 import matplotlib.pyplot as plt
11
12
13 def libsvm_scale_import(filename):
14     """
15     Read data from a libsvm .scale file
16     """
17     datafile = open(filename, 'r')
18
19     # First pass: get dimensions of data
20     num_samples = 0
21     max_feature_id = 0
22     for line in datafile:
23         num_samples += 1
24         tokens = line.split()
25         for feature in tokens[1:]:
26             feature_id = int(feature.split(':')[0])
27             max_feature_id = max(feature_id, max_feature_id)
28
29     # Second pass: read data into array
30     data = np.zeros((num_samples, max_feature_id + 1))
31     curr_sample = 0
32     datafile.seek(0)
33     for line in datafile:
34         tokens = line.split()
35         data[curr_sample][0] = float(tokens[0])
36         for feature in tokens[1:]:
37             feature_id = int(feature.split(':')[0])

```

```

38         feature_val = float(feature.split(':')[1])
39         data[curr_sample][feature_id] = feature_val
40         curr_sample += 1
41     datafile.close()
42
43     print('LOADED:', filename, ':', data.shape)
44
45     return data
46
47
48 def get_neighbors(data, test_sample, num_neighbors):
49     """
50     Given training data, a test sample, and a number of
51     neighbors, return the closest neighbors.
52     """
53     # Calculate all distances from the training samples
54     # to this test sample. Collect index, distance into a list.
55     indices_and_distances = list()
56     for i in range(len(data)):
57         dist = np.linalg.norm(test_sample[1:] - (data[i])[1:]) # leave out classification at
            pos 0
58         indices_and_distances.append([i, dist])
59
60     # Sort list by distance
61     indices_and_distances.sort(key=lambda _: _[1])
62
63     # Make a list of requested number of closest neighbors from sorted
64     # list of indices+distances
65     neighbors = list()
66     for i in range(num_neighbors):
67         neighbors.append(indices_and_distances[i][0])
68
69     return neighbors
70
71
72 def classify_one_sample(data, test_sample, num_neighbors):
73     """
74     Given training data, a test sample, and a number of neighbors,
75     predict which classification the test sample belongs to.
76     """
77     # Get closest neighbors
78     neighbors = get_neighbors(data, test_sample, num_neighbors)
79
80     # Create list of classifications of the neighbors
81     classifications = list()
82     for i in range(len(neighbors)):
83         classifications.append(data[neighbors[i]][0]) # 0 = classification
84
85     # Return the most common classification of the neighbors
86     prediction = max(set(classifications), key = classifications.count)
87     return prediction
88
89
90 def k_nearest_neighbors(data, test_samples, num_neighbors):
91     """
92     Given sample data (samples are rows, columns
93     features, and samples have classifications in position 0),
94     test data, and a number of neighbors, predict which classification
95     each test sample belongs to.
96     """
97     classifications = list()
98     for i in range(len(test_samples)):
99         output = classify_one_sample(data, test_samples[i], num_neighbors)
100         classifications.append(output)
101         if ((i % 20) == 0):
102             print('\rknn test sample', i, end='')
103     print()
104

```

```

105     return(classifications)
106
107
108 def check_knn_classifications(y, y_hat):
109     """
110     Given actual values y and classifications y_hat,
111     return the number of errors
112     """
113     errors = 0
114     for i in range(len(y)):
115         if (y[i] != y_hat[i]):
116             errors += 1
117
118     return errors
119
120
121 def train_perceptron(data, beta, step_limit):
122     """
123     Perceptron. Given a set of data (samples are rows, columns
124     features, and samples have classifications in position 0),
125     a step size (beta), and a step limit, train and return a
126     weight vector that can be used to classify the given data.
127     """
128
129     # Initialize the weight vector including bias element
130     w = np.zeros(len(data[0]))
131
132     # Initialize y_hat
133     y_hat = np.zeros(len(data))
134
135     # Slice off y
136     y = data[:,0]
137
138     # Repeat the main loop until we have convergence or reach the
139     # iteration limit
140     steps = 0
141     converged = False
142     while(not(converged) and (steps < step_limit)):
143         converged = True
144
145         # For each sample in the data, calculate w's classification error
146         # and update w.
147         for i in range(len(data)):
148             # Replace classification in sample[0] with a 1 to allow
149             # for a biased weight vector
150             biased_sample = np.copy(data[i])
151             biased_sample[0] = 1
152
153             # Get prediction and error, then update weight vector
154             y_hat[i] = 1 if (np.matmul(w.T, biased_sample) > 0) else -1
155             error = y[i] - y_hat[i]
156             w += biased_sample * error * beta
157             steps += 1
158
159             # If error on this element is > a very small value, we have
160             # not converged.
161             if (abs(error) > 0.000001):
162                 converged = False
163
164     print('Perceptron: ', steps, 'steps; converged?', converged)
165
166     return w
167
168
169 def multiclass_train_perceptron(data, beta, step_limit):
170     """
171     Perceptron. Given a set of data (samples are rows, columns
172     features, and samples have classifications in position 0),

```



```

173     a step size (beta), and a step limit, train and return a
174     weight vector that can be used to classify the given data.
175
176     This version works on data with multiple classes by one-vs-rest.
177     """
178     # Find unique classes
179     classes = []
180     for i in range(data.shape[0]):
181         if (not(data[i][0] in classes)):
182             classes.append(data[i][0])
183
184     # For each classification, train perceptron on current class vs.
185     # rest of the untrained classes.
186     ws = []
187     curr_data = copy.deepcopy(data)
188     for curr_class in range(len(classes)):
189
190         # Save original classification data
191         orig_classes = copy.deepcopy(curr_data[:,0])
192
193         # Reset classification data to 1 (for current class) or -1 for other
194         for i in range(curr_data.shape[0]):
195             if (curr_data[i][0] == classes[curr_class]):
196                 curr_data[i][0] = 1
197             else:
198                 curr_data[i][0] = -1
199
200         # Train and find weights
201         ws.append(train_perceptron(curr_data, beta, step_limit))
202
203         # Put original classifications back
204         for i in range(curr_data.shape[0]):
205             curr_data[i][0] = orig_classes[i]
206
207     return ws
208
209
210 def test_perceptron(data, w):
211     """
212     Given test data and a weight vector w, return number of
213     num_misclass when classifying the test data using the
214     weights.
215     """
216     errors = 0
217
218     # Initialize y_hat
219     y_hat = np.zeros(len(data))
220
221     # Slice off y
222     y = data[:,0]
223
224     # Determine how weights classify each test sample and count
225     # num_misclass
226     for i in range(len(data)):
227         biased_sample = np.copy(data[i])
228         biased_sample[0] = 1
229         y_hat[i] = 1 if (np.matmul(w.T, biased_sample) > 0) else -1
230         if (y[i] != y_hat[i]):
231             errors += 1
232
233     return errors
234
235
236 def multiclass_test_perceptron(data, ws):
237     """
238     Given test data and a weight vector w, return number of
239     num_misclass when classifying the test data using the
240     weights.

```

```

241
242 This version works on data with multiple classes by One vs. All (OVA).
243 """
244 # Find unique classes
245 classes = []
246 for i in range(data.shape[0]):
247     if (not(data[i][0] in classes)):
248         classes.append(data[i][0])
249
250 # For each classification, test perceptron on current class vs.
251 # rest of the untested classes.
252 errors = []
253 curr_data = copy.deepcopy(data)
254 for curr_class in range(len(classes)):
255
256     # Save original classification data
257     orig_classes = copy.deepcopy(curr_data[:,0])
258
259     # Reset classification data to 1 (for current class) or -1 for other
260     for i in range(curr_data.shape[0]):
261         if (curr_data[i][0] == classes[curr_class]):
262             curr_data[i][0] = 1
263         else:
264             curr_data[i][0] = -1
265
266     # Train and find weights
267     errors.append(test_perceptron(curr_data, ws[curr_class]))
268
269     # Put original classifications back
270     for i in range(curr_data.shape[0]):
271         curr_data[i][0] = orig_classes[i]
272
273 return errors
274
275
276 def iris_knn():
277     """
278     Run kNN on the iris dataset for the various numbers of neighbors.
279     """
280     print("—————\niris kNN")
281
282     # Load data
283     data = libsvm.scale-import('data/iris.scale')
284
285     # Shuffle the data because we want to split it into train & test,
286     # and it is pre-sorted (we would test against classes we didn't
287     # see in training)
288     np.random.seed(1) # ensure consistent shuffling
289     np.random.shuffle(data)
290
291     # Split up data into training and test data based on split value
292     split = 50
293     train_data = data[:,split]
294     test_data = data[split:]
295
296     # Test multiple values of k
297     test_ks = np.arange(1, split)
298     error_rates = np.zeros(test_ks.shape[0])
299     for i in range(len(test_ks)):
300         # Classify the test data
301         print('Classify with k =', test_ks[i])
302         classifications = k_nearest_neighbors(train_data, test_data,
303                                             test_ks[i])
304
305         # Check accuracy
306         errors = check_knn_classifications(test_data[:,0], classifications)
307         error_rates[i] = errors / test_data.shape[0]
308         print(errors, 'errors in', test_data.shape[0], 'samples')

```

```

309     print('ks:', test_ks)
310     print('error rates:', error_rates)
311     plt.clf()
312     plt.plot(test_ks, error_rates, marker='.')
313     plt.title('Iris kNN: error rate vs. k')
314     plt.xlabel('k')
315     plt.ylabel('error rate')
316     plt.xlim(left = 0)
317     plt.ylim(bottom = 0)
318     plt.grid(True)
319     plt.savefig('iris-knn.png', dpi = 600)
320
321
322     def iris_perceptron():
323         """
324         Run Perceptron on the iris dataset in various ways.
325         """
326         print("—————\niris Perceptron")
327
328         # Load data
329         data = libsvm.scale_import('data/iris.scale')
330
331         # Shuffle the data because we want to split it into train & test,
332         # and it is pre-sorted (we would test against classes we didn't
333         # see in training)
334         np.random.seed(1) # ensure consistent shuffling
335         np.random.shuffle(data)
336
337         # Split up data into training and test data based on split value
338         split = 50
339         train_data = data[:split]
340         test_data = data[split:]
341
342         # Perform multi-class training and test and collect
343         # a weight vector and number of errors for each class
344         ws = multiclass_train_perceptron(train_data, 0.1, 100000)
345         errors = multiclass_test_perceptron(test_data, ws)
346
347         # Report errors
348         print(errors, 'errors in', test_data.shape[0], 'samples')
349
350         # Show sorted weights for every class
351         for i in range(len(ws)):
352
353             # Sort weights to find most important
354             w = list(ws[i][1:])
355             feature_ids = range(1, len(w) + 1)
356             print('W:', w)
357             labels = []
358             for id in feature_ids:
359                 labels.append(str(int(id)))
360
361             # Report top weights
362             plt.clf()
363             plt.bar(labels, w)
364             plt.title('iris Perceptron: feature weights for class = ' + str(i+1))
365             plt.xlabel('feature ID')
366             plt.ylabel('weight')
367             plt.grid(True)
368             plt.savefig('iris-weights' + str(i+1) + '.png', dpi = 600)
369
370
371     def a4a_knn():
372         """
373         Run kNN on the a4a dataset for various numbers of neighbors.
374         """
375         print("—————\na4a kNN")
376

```

```

377 # Load data
378 train_data = libsvm_scale_import('data/a4a')
379 test_data = libsvm_scale_import('data/a4a.t')
380
381 # Training data has 1 fewer feature than test data, so add a column
382 # of zeros to it so samples have same number of features in train and test
383 zero_col = np.zeros((len(train_data), 1))
384 train_data = np.hstack((train_data, zero_col))
385
386 # Test multiple values of k
387 # This takes over 3 hours to run on my fastest computer.
388 test_ks = np.array([1, 3, 5, 11, 21, 31, 41, 51, 61, 71, 81, 91, 101, 201, 301, 401,
389 501, 601, 701, 801, 901, 1001])
390 error_rates = np.zeros(len(test_ks))
391 for i in range(len(test_ks)):
392     print('Classify with k =', test_ks[i])
393     # Classify the test data
394     classifications = k_nearest_neighbors(train_data, test_data,
395                                           test_ks[i])
396     # Check accuracy
397     errors = check_knn_classifications(test_data[:,0], classifications)
398     error_rates[i] = errors / test_data.shape[0]
399     print(errors, 'errors in', test_data.shape[0], 'samples')
400
401 print('ks:', test_ks)
402 print('error rates:', error_rates)
403 plt.clf()
404 plt.plot(test_ks, error_rates, marker='.')
405 plt.title('a4a kNN: error rate vs. k')
406 plt.xlabel('k')
407 plt.ylabel('error rate')
408 plt.xlim(left = 0)
409 plt.ylim(bottom = 0)
410 plt.grid(True)
411 plt.savefig('a4a_knn.png', dpi = 600)
412
413 def a4a_perceptron():
414     """
415     Run Perceptron on the a4a dataset in various ways.
416     """
417     print("—————\na4a Perceptron")
418
419     # Load data
420     train_data = libsvm_scale_import('data/a4a')
421     test_data = libsvm_scale_import('data/a4a.t')
422
423     # Training data has 1 fewer feature than test data, so add a column
424     # of zeros to it so samples have same number of features in train and test
425     zero_col = np.zeros((len(train_data), 1))
426     train_data = np.hstack((train_data, zero_col))
427
428     # Test multiple values of beta
429     test_betas = np.array([0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0])
430     error_rates = np.zeros(test_betas.shape[0])
431     ws = []
432     best_beta = -1
433     best_error_rate = 999999
434     for i in range(len(test_betas)):
435         print('Classify with beta =', test_betas[i])
436
437         # Train and find weights
438         ws.append(train_perceptron(train_data, test_betas[i], 100000))
439
440         # Check accuracy
441         errors = test_perceptron(test_data, ws[i])
442         error_rates[i] = errors / test_data.shape[0]
443         if (error_rates[i] < best_error_rate):

```

```

444         best_error_rate = error_rates[i]
445         best_beta = i
446         print(errors, 'errors in', test_data.shape[0], 'samples')
447
448     # Report error rates
449     print('betas:', test_betas)
450     print('error rates:', error_rates)
451     plt.clf()
452     plt.plot(test_betas, error_rates, marker='.')
453     plt.title('a4a Perceptron: error rate vs. step size for 100000 iterations')
454     plt.xscale('log')
455     plt.xlabel('step size')
456     plt.ylabel('error rate')
457     plt.ylim(bottom = 0)
458     plt.grid(True)
459     plt.savefig('a4a-perceptron.png', dpi = 600)
460
461     # Sort weights to find most important
462     w = list(ws[best_beta][1:])
463     feature_ids = range(1, len(w) + 1)
464     bar_data = list(zip(feature_ids, w))
465     bar_data.sort(key = lambda -: abs(-[1]), reverse = True)
466     bar_data = np.array(bar_data[:20])
467     labels = []
468     for id in bar_data[:,0]:
469         labels.append(str(int(id)))
470
471     # Report top weights
472     plt.clf()
473     plt.bar(labels, bar_data[:,1])
474     plt.title('a4a Perceptron: 20 most important features')
475     plt.xlabel('feature ID')
476     plt.ylabel('weight')
477     plt.grid(True)
478     plt.savefig('a4a-weights.png', dpi = 600)
479
480
481 def main():
482     iris_knn()
483     iris_perceptron()
484     a4a_knn()
485     a4a_perceptron()
486
487
488 if __name__ == '__main__':
489     main()

```

MiniProj1.py

7 Run Log

```

1
2 iris knn
3 LOADED: data/iris.scale : (150, 5)
4 Classify with k = 1
5 knn test sample 80
6 6 errors in 100 samples
7 Classify with k = 2
8 knn test sample 80
9 12 errors in 100 samples
10 Classify with k = 3
11 knn test sample 80
12 8 errors in 100 samples
13 Classify with k = 4
14 knn test sample 80
15 9 errors in 100 samples

```

```

16 Classify with k = 5
17 knn test sample 80
18 7 errors in 100 samples
19 Classify with k = 6
20 knn test sample 80
21 11 errors in 100 samples
22 Classify with k = 7
23 knn test sample 80
24 8 errors in 100 samples
25 Classify with k = 8
26 knn test sample 80
27 12 errors in 100 samples
28 Classify with k = 9
29 knn test sample 80
30 12 errors in 100 samples
31 Classify with k = 10
32 knn test sample 80
33 12 errors in 100 samples
34 Classify with k = 11
35 knn test sample 80
36 10 errors in 100 samples
37 Classify with k = 12
38 knn test sample 80
39 13 errors in 100 samples
40 Classify with k = 13
41 knn test sample 80
42 12 errors in 100 samples
43 Classify with k = 14
44 knn test sample 80 40
45 14 errors in 100 samples
46 Classify with k = 15
47 knn test sample 80
48 12 errors in 100 samples
49 Classify with k = 16
50 knn test sample 80
51 12 errors in 100 samples
52 Classify with k = 17
53 knn test sample 80
54 12 errors in 100 samples
55 Classify with k = 18
56 knn test sample 80
57 14 errors in 100 samples
58 Classify with k = 19
59 knn test sample 80
60 14 errors in 100 samples
61 Classify with k = 20
62 knn test sample 80
63 16 errors in 100 samples
64 Classify with k = 21
65 knn test sample 80
66 15 errors in 100 samples
67 Classify with k = 22
68 knn test sample 80
69 19 errors in 100 samples
70 Classify with k = 23
71 knn test sample 80
72 16 errors in 100 samples
73 Classify with k = 24
74 knn test sample 80
75 19 errors in 100 samples
76 Classify with k = 25
77 knn test sample 80
78 18 errors in 100 samples
79 Classify with k = 26
80 knn test sample 80
81 20 errors in 100 samples
82 Classify with k = 27
83 knn test sample 80

```

```

84 19 errors in 100 samples
85 Classify with k = 28
86 knn test sample 80
87 36 errors in 100 samples
88 Classify with k = 29
89 knn test sample 80
90 36 errors in 100 samples
91 Classify with k = 30
92 knn test sample 80
93 36 errors in 100 samples
94 Classify with k = 31
95 knn test sample 80
96 36 errors in 100 samples
97 Classify with k = 32
98 knn test sample 80 40
99 36 errors in 100 samples
100 Classify with k = 33
101 knn test sample 80
102 36 errors in 100 samples
103 Classify with k = 34
104 knn test sample 80
105 36 errors in 100 samples
106 Classify with k = 35
107 knn test sample 80
108 67 errors in 100 samples
109 Classify with k = 36
110 knn test sample 80
111 69 errors in 100 samples
112 Classify with k = 37
113 knn test sample 80
114 69 errors in 100 samples
115 Classify with k = 38
116 knn test sample 80
117 69 errors in 100 samples
118 Classify with k = 39
119 knn test sample 80
120 69 errors in 100 samples
121 Classify with k = 40
122 knn test sample 60 80
123 69 errors in 100 samples
124 Classify with k = 41
125 knn test sample 80
126 69 errors in 100 samples
127 Classify with k = 42
128 knn test sample 80
129 69 errors in 100 samples
130 Classify with k = 43
131 knn test sample 80
132 69 errors in 100 samples
133 Classify with k = 44
134 knn test sample 80
135 69 errors in 100 samples
136 Classify with k = 45
137 knn test sample 80
138 69 errors in 100 samples
139 Classify with k = 46
140 knn test sample 80
141 69 errors in 100 samples
142 Classify with k = 47
143 knn test sample 80
144 69 errors in 100 samples
145 Classify with k = 48
146 knn test sample 8020
147 69 errors in 100 samples
148 Classify with k = 49
149 knn test sample 80
150 69 errors in 100 samples
151 ks: [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

```

```

152 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
153 49]
154 error rates: [0.06 0.12 0.08 0.09 0.07 0.11 0.08 0.12 0.12 0.12 0.1 0.13 0.12 0.14
155 0.12 0.12 0.12 0.14 0.14 0.16 0.15 0.19 0.16 0.19 0.18 0.2 0.19 0.36
156 0.36 0.36 0.36 0.36 0.36 0.36 0.67 0.69 0.69 0.69 0.69 0.69 0.69
157 0.69 0.69 0.69 0.69 0.69 0.69 0.69]
158
159 iris Perceptron
160 LOADED: data/iris.scale : (150, 5)
161 Perceptron: 100 steps; converged? True
162 Perceptron: 100000 steps; converged? False
163 Perceptron: 250 steps; converged? True
164 [0, 68, 58] errors in 100 samples
165 W: [-0.1222222, 0.15, -0.28813540000000004, -0.2666666]
166 W: [-0.16614598199997474, -0.46739085999993474, 0.5187340000001649, -0.6666498142802026]
167 W: [0.05555557298200006, -0.33333285999999995, 0.39322, 0.35000042421438]
168
169 a4a knn
170 LOADED: data/a4a : (4781, 123)
171 LOADED: data/a4a.t : (27780, 124)
172 Classify with k = 1
173 knn test sample 27760
174 5937 errors in 27780 samples
175 Classify with k = 3
176 knn test sample 27760
177 5238 errors in 27780 samples
178 Classify with k = 5
179 knn test sample 27760
180 5028 errors in 27780 samples
181 Classify with k = 11
182 knn test sample 27760
183 4769 errors in 27780 samples
184 Classify with k = 21
185 knn test sample 27760
186 4663 errors in 27780 samples
187 Classify with k = 31
188 knn test sample 27760
189 4639 errors in 27780 samples
190 Classify with k = 41
191 knn test sample 27760
192 4626 errors in 27780 samples
193 Classify with k = 51
194 knn test sample 27760
195 4664 errors in 27780 samples
196 Classify with k = 61
197 knn test sample 27760
198 4656 errors in 27780 samples
199 Classify with k = 71
200 knn test sample 27760
201 4668 errors in 27780 samples
202 Classify with k = 81
203 knn test sample 27760
204 4695 errors in 27780 samples
205 Classify with k = 91
206 knn test sample 27760
207 4714 errors in 27780 samples
208 Classify with k = 101
209 knn test sample 27760
210 4721 errors in 27780 samples
211 Classify with k = 201
212 knn test sample 27760
213 4845 errors in 27780 samples
214 Classify with k = 301
215 knn test sample 27760
216 4871 errors in 27780 samples
217 Classify with k = 401
218 knn test sample 27760
219 4911 errors in 27780 samples

```



```

220 Classify with k = 501
221 knn test sample 27760
222 4940 errors in 27780 samples
223 Classify with k = 601
224 knn test sample 27760
225 4980 errors in 27780 samples
226 Classify with k = 701
227 knn test sample 27760
228 5007 errors in 27780 samples
229 Classify with k = 801
230 knn test sample 27760
231 5052 errors in 27780 samples
232 Classify with k = 901
233 knn test sample 27760
234 5109 errors in 27780 samples
235 Classify with k = 1001
236 knn test sample 27760
237 5227 errors in 27780 samples
238 ks: [ 1 3 5 11 21 31 41 51 61 71 81 91 101 201
239 301 401 501 601 701 801 901 1001]
240 error rates: [0.2137149 0.18855292 0.18099352 0.17167027 0.16785457 0.16699064
241 0.16652268 0.16789057 0.16760259 0.16803456 0.16900648 0.16969042
242 0.1699424 0.17440605 0.17534197 0.17678186 0.17782577 0.17926566
243 0.18023758 0.18185745 0.18390929 0.18815695]
244
245 a4a Perceptron
246 LOADED: data/a4a : (4781, 123)
247 LOADED: data/a4a.t : (27780, 124)
248 Classify with beta = 0.0001
249 Perceptron: 100401 steps; converged? False
250 5094 errors in 27780 samples
251 Classify with beta = 0.001
252 Perceptron: 100401 steps; converged? False
253 5911 errors in 27780 samples
254 Classify with beta = 0.01
255 Perceptron: 100401 steps; converged? False
256 7966 errors in 27780 samples
257 Classify with beta = 0.1
258 Perceptron: 100401 steps; converged? False
259 5450 errors in 27780 samples
260 Classify with beta = 1.0
261 Perceptron: 100401 steps; converged? False
262 6052 errors in 27780 samples
263 Classify with beta = 10.0
264 Perceptron: 100401 steps; converged? False
265 6052 errors in 27780 samples
266 Classify with beta = 100.0
267 Perceptron: 100401 steps; converged? False
268 6052 errors in 27780 samples
269 betas: [1.e-04 1.e-03 1.e-02 1.e-01 1.e+00 1.e+01 1.e+02]
270 error rates: [0.18336933 0.21277898 0.28675306 0.19618431 0.21785457 0.21785457
271 0.21785457]

```
