1. Assume that data is linearly separable by a hyperplane through the origin. Run two, three and four passes of perceptron, voted perceptron, and averaged perceptron on the training dataset to find classifiers that separate the two classes. What are the training errors and the test errors of perceptron, voted perceptron and averaged perceptron after two, three and four passes?

After two passes:

	Perceptron	Voted Perceptron	Averaged Perceptron
Training Error	0.0404	0.0404	0.0541
Test Error	0.0584	0.0610	0.0822

After three passes:

	Perceptron	Voted Perceptron	Averaged Perceptron
Training Error	0.0211	0.0303	0.0376
Test Error	0.0451	0.0451	0.0610

After four passes:

	Perceptron	Voted Perceptron	Averaged Perceptron
Training Error	0.0183	0.0248	0.0339
Test Error	0.0477	0.0451	0.0504

2. Find the three coordinates in w_{avg} with the highest and lowest values. What are the words (from pa3dictionary.txt) that correspond to these coordinates? The three highest coordinates are those words whose presence indicates the positive class most strongly, and the three lowest coordinates are those words whose presence indicates the negative class most strongly.

Note: we map label 1 to +1 and label 2 to −1.

The words that correspond to the three highest coordinates (in decreasing order) are “file”, “program”, and “line” with “file” having the most positive coordinate value.

The words that correspond to the three lowest coordinates (in increasing order) are “he”, “team”, and “game” with “he” having the most negative coordinate value.

3. Write down the confusion matrix for the one-vs-all classifier on the training data in pa3train.txt based on the test data in pa3test.txt. Looking at the confusion matrix, what are the i and j in the following statements?

The confusion matrix:

```
[[0.71891892 0.00520833 0.03428571 0.02173913 0.         0.        ]
 [0.01081081 0.65625    0.03428571 0.02717391 0.01282051 0.01851852]
 [0.         0.015625   0.37142857 0.         0.         0.02777778]
 [0.01621622 0.00520833 0.         0.69021739 0.         0.        ]
 [0.01621622 0.03125    0.07428571 0.00543478 0.80128205 0.12037037]
 [0.00540541 0.01041667 0.03428571 0.         0.07051282 0.49074074]
 [0.23243243 0.27604167 0.45142857 0.25543478 0.11538462 0.34259259]]
```

(a). The perceptron classifier has the highest accuracy for examples that belong to class i.

According to the matrix, the perceptron classifier has the highest accuracy for examples that belong to class 5.

(b). The perceptron classifier has the least accuracy for examples that belong to class i.

According to the matrix, the perceptron classifier has the least accuracy for examples that belong to class 3.

(c). The perceptron classifier most often mistakenly classifies an example in class j as belonging to class i, for i, j ∈ {1, 2, 3, 4, 5, 6} (i.e., excluding Don't Know).

According to the matrix, the perceptron classifier **most** often mistakenly classifies an example in class 6 as belonging to class 5.

For examples in other classes:
An example in class 1 is often mistakenly classified as belonging to either class 4 or class 5.
An example in class 2 is often mistakenly classified as belonging to class 5.
An example in class 3 is often mistakenly classified as belonging to class 5.
An example in class 4 is often mistakenly classified as belonging to class 2.
An example in class 5 is often mistakenly classified as belonging to class 6.

pa3

May 17, 2018

In [ ]:
```
import numpy
import copy
import heapq
import random

class Solver():
    def __init__(self):
        # list that stores all the training data
        self.training_data = []
        # list that stores all the test data
        self.test_data = []
        # list that stores all the words
        self.dictionary = []
        # list that stores all the training data with label 1 and 2
        self.training_1_2 = []
        # list that stores all the test data with label 1 and 2
        self.test_1_2 = []
    def load_data(self):
        # open the training data file
        training_file = open("pa3train.txt")
        # load the training data
        for line in training_file:
            data = line.split()
            for i in range(0, len(data)):
                data[i] = float(data[i])
            self.training_data.append(data)
        # close the training data file
        training_file.close()
        # open the test data file
        test_file = open("pa3test.txt")
        # load the test data
        for line in test_file:
            data = line.split()
            for i in range(0, len(data)):
                data[i] = float(data[i])
            self.test_data.append(data)
        # close the test data file
```

```
        test_file.close()
        # open the dictionary file
        dictionary_file = open("pa3dictionary.txt")
        # load the words
        for word in dictionary_file:
            self.dictionary.append(word)
        # close the dicitonary file
        dictionary_file.close()
    def load_data_1_2(self):
        # load the corresponding training data
        for i in self.training_data:
            if i[-1] == 1:
                j = copy.deepcopy(i)
                j[-1] = 1
                self.training_1_2.append(j)
            if i[-1] == 2:
                j = copy.deepcopy(i)
                j[-1] = -1
                self.training_1_2.append(j)
        # load the corresponding test data
        for i in self.test_data:
            if i[-1] == 1:
                j = copy.deepcopy(i)
                j[-1] = 1
                self.test_1_2.append(j)
            if i[-1] == 2:
                j = copy.deepcopy(i)
                j[-1] = -1
                self.test_1_2.append(j)
    def perceptron(self, round):
        # initialize the normal vector w
        w = [0] * 819
        w = numpy.array(w)
        p = 0
        while (p < round):
            # iterate through all the training data
            for i in self.training_1_2:
                # if the prediciton is incorrect
                if i[-1] * (numpy.dot(w, numpy.array(i[:-1]))) <= 0:
                    # update w
                    w = w + i[-1] * numpy.array(i[:-1])
            # increment the pass
            p = p + 1
        # calculate the test error
        error = 0
        for i in self.test_1_2:
            prediction = numpy.dot(w, numpy.array(i[0:819]))
            if prediction == 0:
```

```
                '''
                m = [-1, 1]
                r = random.randint(0, 1)
                prediction = m[r]
                '''
                prediction = -1
            # if the prediction is incorrect
            if (prediction > 0 and i[819] == -1) or (prediction < 0 and i[819] == 1):
                error = error + 1
        test_error = float(error) / float(len(self.test_1_2))
        # calculate the training error
        error = 0
        for i in self.training_1_2:
            prediction = numpy.dot(w, numpy.array(i[:-1]))
            if prediction == 0:
                '''
                m = [-1, 1]
                r = random.randint(0, 1)
                prediction = m[r]
                '''
                prediction = -1
            # if the prediciton is incorrect
            if prediction * i[-1] < 0:
                error = error + 1
        training_error = float(error) / float(len(self.training_1_2))
        return test_error, training_error
    def voted_perceptron(self, round):
        # list that stores all the (classifier, count) pairs
        l = []
        # initialize the normal vector w
        w = [0] * 819
        w = numpy.array(w)
        # initialize the count
        c = 1
        p = 0
        while (p < round):
            # pop the classifier and count if list is not empty
            if (len(l) != 0):
                w, c = l.pop()
            # iterate through all the training data
            for i in self.training_1_2:
                # if the prediction is incorrect
                if i[819] * (numpy.dot(w , numpy.array(i[0:819]))) <= 0:
                    # store the (classifier, count) pair to the list
                    l.append((w, c))
                    # update w
                    w = w + i[819] * numpy.array(i[0:819])
                    # set count to 1
```

```
                c = 1
            else:
                # increment the count
                c = c + 1
        l.append((w, c))
        # increment the pass
        p = p + 1
    # calculate the test error
    error = 0
    for i in self.test_1_2:
        s = 0
        # calculate the weighted sum
        for j in l:
            s = s + j[1] * numpy.sign(numpy.dot(j[0], numpy.array(i[0:819])))
        prediction = numpy.sign(s)
        if prediction == 0:
            '''
            m = [-1, 1]
            r = random.randint(0, 1)
            prediction = m[r]
            '''
            prediction = -1
        # if the prediction is incorrect
        if (prediction > 0 and i[819] == -1) or (prediction < 0 and i[819] == 1):
            error = error + 1
    test_error = float(error) / float(len(self.test_1_2))
    # calculate the training error
    error = 0
    for i in self.training_1_2:
        s = 0
        # calculate the weighted sum
        for j in l:
            s = s + j[1] * numpy.sign(numpy.dot(j[0], numpy.array(i[0:819])))
        prediction = numpy.sign(s)
        if prediction == 0:
            '''
            m = [-1, 1]
            r = random.randint(0, 1)
            prediction = m[r]
            '''
            prediction = -1
        # if the prediction is incorrect
        if (prediction > 0 and i[819] == -1) or (prediction < 0 and i[819] == 1):
            error = error + 1
    training_error = float(error) / float(len(self.training_1_2))
    return test_error, training_error
def averaged_perceptron(self, round):
    # initialize the sum
```

```
s = 0
# initialize the normal vector w
w = [0] * 819
w = numpy.array(w)
# initialize the count
c = 1
p = 0
while (p < round):
    # iterate through all the training data
    for i in self.training_1_2:
        # if the prediction is incorrect
        if i[819] * (numpy.dot(w , numpy.array(i[0:819]))) <= 0:
            # update the sum
            s = s + c * w
            # update w
            w = w + i[819] * numpy.array(i[0:819])
            # set count to 1
            c = 1
        else:
            # increment the count
            c = c + 1
    s = s + c * w
    # increment the pass
    p = p + 1
# calculate the test error
error = 0
for i in self.test_1_2:
    prediction = numpy.sign(numpy.dot(s, numpy.array(i[0:819])))
    if prediction == 0:
        '''
        m = [-1, 1]
        r = random.randint(0, 1)
        prediction = m[r]
        '''
        prediction = -1
    # if the prediction is incorrect
    if (prediction > 0 and i[819] == -1) or (prediction < 0 and i[819] == 1):
        error = error + 1
test_error = float(error) / float(len(self.test_1_2))
# calculate the training error
error = 0
for i in self.training_1_2:
    prediction = numpy.sign(numpy.dot(s, numpy.array(i[0:819])))
    if prediction == 0:
        '''
        m = [-1, 1]
        r = random.randint(0, 1)
        prediction = m[r]
```

```
                '''
                prediction = -1
            # if the prediction is incorrect
            if (prediction > 0 and i[819] == -1) or (prediction < 0 and i[819] == 1):
                error = error + 1
        training_error = float(error) / float(len(self.training_1_2))
        # return the classifier as a list
        l = list(s)
        return test_error, training_error, l
    def find_coordinates(self, l):
        # push the coordinates into the heap
        heap = []
        for i in l:
            heapq.heappush(heap, i)
        ordered = []
        # pop from the heap
        while heap:
            ordered.append(heapq.heappop(heap))
        smallest = (ordered[0], ordered[1], ordered[2])
        largest = (ordered[-1], ordered[-2], ordered[-3])
        smallest_coordinates = [0] * 3
        largest_coordinates = [0] * 3
        # find the coordinates
        for i in range(0, len(l)):
            if l[i] == smallest[0]:
                smallest_coordinates[0] = i
            if l[i] == smallest[1]:
                smallest_coordinates[1] = i
            if l[i] == smallest[2]:
                smallest_coordinates[2] = i
            if l[i] == largest[0]:
                largest_coordinates[0] = i
            if l[i] == largest[1]:
                largest_coordinates[1] = i
            if l[i] == largest[2]:
                largest_coordinates[2] = i
        # find the words
        print("words with smallest values")
        for i in smallest_coordinates:
            print(self.dictionary[i])
        print("words with largest values")
        for i in largest_coordinates:
            print(self.dictionary[i])
    def one_vs_all(self, i):
        # class i vs other classes
        new_training_data = []
        for j in self.training_data:
            if j[-1] == i:
```

```
                k = copy.deepcopy(j)
                k[-1] = 1
                new_training_data.append(k)
            else:
                k = copy.deepcopy(j)
                k[-1] = -1
                new_training_data.append(k)
        # use perceptron to generate the classifier
        w = [0] * 819
        w = numpy.array(w)
        for j in new_training_data:
            if j[-1] * (numpy.dot(w, numpy.array(j[:-1]))) <= 0:
                w = w + j[-1] * numpy.array(j[:-1])
        # return the classifier
        return w
    def build_matrix(self, w_list):
        n_list = [0] * 7
        for i in self.test_data:
            n_list[int(i[-1])] = n_list[int(i[-1])] + 1
        # generate a 8 x 7 matrix
        matrix = numpy.zeros((8, 7))
        # iterate through all the test data
        for i in self.test_data:
            # get the label of the test data
            label = int(i[-1])
            # list that contains the predictions
            l = []
            # iterate through all the classifiers
            for w in w_list:
                l.append(numpy.sign(numpy.dot(w, numpy.array(i[:-1]))))
            prediction = -1
            count = 0
            for j in range(0, len(l)):
                if l[j] == 1:
                    count = count + 1
                    prediction = j + 1
            # update the matrix
            if count != 1:
                matrix[7, label] = matrix[7, label] + 1
            else:
                matrix[prediction, label] = matrix[prediction, label] + 1
        # iterate through the matrix
        for r in range(1, 8):
            for c in range(1, 7):
                matrix[r, c] = float(matrix[r, c]) / float(n_list[c])
        print(matrix[1:, 1:])

if __name__ == '__main__':
```

```
solver = Solver()
solver.load_data()
solver.load_data_1_2()
# Single Passes
print("Single Passes:")
print("Perceptron")
test_error, training_error = solver.perceptron(1)
print("Test Error:", test_error, "Training Error:", training_error)
print("Voted Perceptron")
test_error, training_error = solver.voted_perceptron(1)
print("Test Error:", test_error, "Training Error:", training_error)
print("Averaged Perceptron")
test_error, training_error, l = solver.averaged_perceptron(1)
print("Test Error:", test_error, "Training Error:", training_error)
print("\n")
# Two Passes
print("Two Passes:")
print("Perceptron")
test_error, training_error = solver.perceptron(2)
print("Test Error:", test_error, "Training Error:", training_error)
print("Voted Perceptron")
test_error, training_error = solver.voted_perceptron(2)
print("Test Error:", test_error, "Training Error:", training_error)
print("Averaged Perceptron")
test_error, training_error, l = solver.averaged_perceptron(2)
print("Test Error:", test_error, "Training Error:", training_error)
print("\n")
# Three Passes
print("Three Passes:")
print("Perceptron")
test_error, training_error = solver.perceptron(3)
print("Test Error:", test_error, "Training Error:", training_error)
print("Voted Perceptron")
test_error, training_error = solver.voted_perceptron(3)
print("Test Error:", test_error, "Training Error:", training_error)
print("Averaged Perceptron")
test_error, training_error, l = solver.averaged_perceptron(3)
print("Test Error:", test_error, "Training Error:", training_error)
# find the coordinates
solver.find_coordinates(l)
print("\n")
# Four Passes
print("Four Passes:")
print("Perceptron")
test_error, training_error = solver.perceptron(4)
print("Test Error:", test_error, "Training Error:", training_error)
print("Voted Perceptron")
test_error, training_error = solver.voted_perceptron(4)
```

```
print("Test Error:", test_error, "Training Error:", training_error)
print("Averaged Perceptron")
test_error, training_error, l = solver.averaged_perceptron(4)
print("Test Error:", test_error, "Training Error:", training_error)
# build the confusion matrix
w_list = []
for i in range(1, 7):
    w_list.append(solver.one_vs_all(i))
solver.build_matrix(w_list)
```