CS228: Human Computer Interaction

Deliverable 4

Due: Monday, September 29, 11:59pm

Description

In this and the next deliverable, you are going to develop a third Python program. This program will implement a machine learning algorithm that takes as input a user's hand gesture and predicts which ASL letter (if any) the user attempted to sign. Before we get to that however, in this current deliverable you will learn to work with a common machine learning algorithm called K Nearest Neighbors. In the next deliverable, we will apply this algorithm to the hand gestures you recorded in Deliverable 3. **Note:** You will not need your Leap Motion device for this deliverable.

1. In this deliverable you will work with three Python libraries:

   (a) Matplotlib, which you'll use to visualize the predictions of your algorithm.

   (b) Numpy, which allows you to work with vectors and matrices.

   (c) Scikit-learn, a collection of machine learning algorithms in Python.

   You have already used the first two libraries. If you installed Canopy, you will already have Scikit-learn. If you did not, you will need to install Scikit-learn yourself: instructions are here.

2. Start by creating a new Python program called `Predict.py` with the single line

   (a) `print 1`

   Run it.

3. At the top of the program, include the three libraries:

   (a) `import numpy as np`

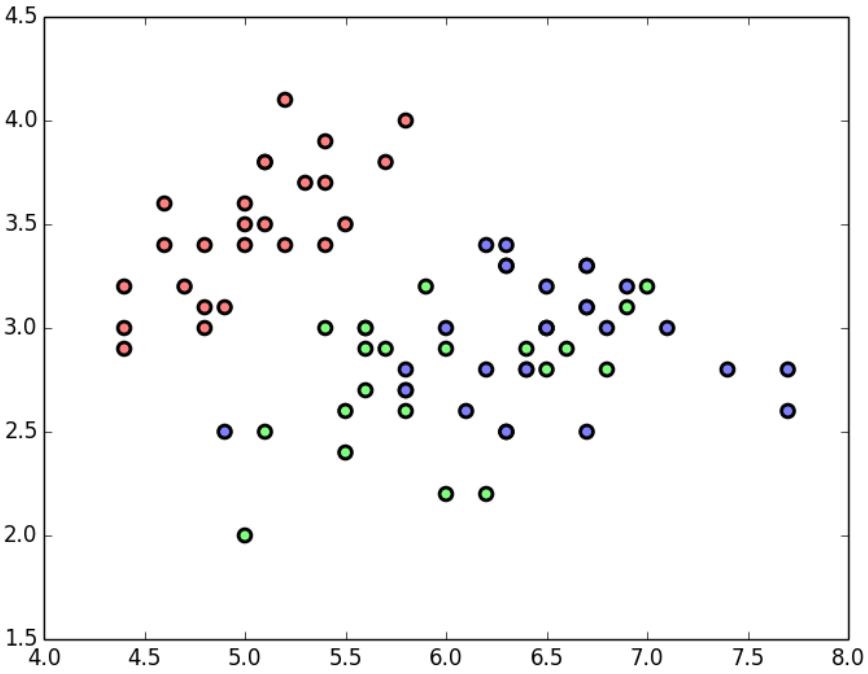
   (b) `import matplotlib.pyplot as plt`

   (c) `from sklearn import datasets`

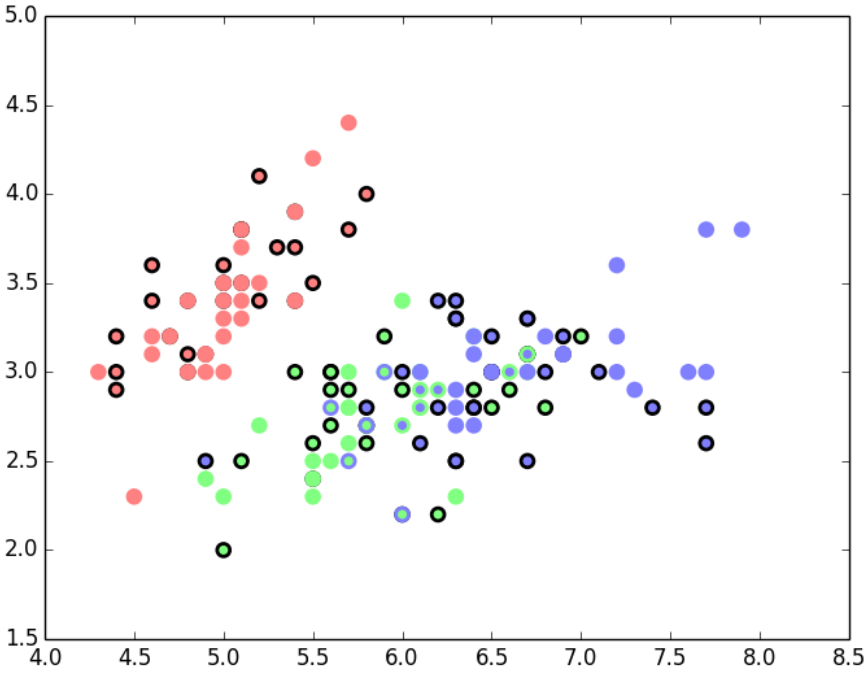
   The `datasets` variable contains a number of data sets that we will use to set up our machine learning algorithm and make sure that it works correctly.

4. Replace line 2(a) with

   (a) `iris = datasets.load_iris()`



a



b

Figure 1: (a): The original training points. (b) The training points as well as the testing points.

This will load the iris data set, which is a common data set used to test out machine learning algorithms. In short, this data set contains observations of 150 flowers. For each flower, four features that describe that flower were measured, along with the particular species of Iris to which that flower belonged. Please read just the introductory text about this data set on its Wikipedia page.

5. Add `print iris.data` after you load the data. Now when you run your program you should see a matrix printed with 150 rows (corresponding to each of the observed flowers) and four columns (corresponding to the four measured features for that flower).

6. Change this last line to `print iris.data[:,0:2]`. This will print out all of the rows (indicated by the colon before the comma) and the first two columns (columns zero up to but not including column two) of this data. When you run your program now you should see a 150-by-two matrix printed.

7. **Note:** Throughout this deliverable we are going to *slice* this matrix in various ways. If you get confused, it is worthwhile to make your way through this short Python tutorial which will introduce you to slicing. Although the tutorial uses strings rather than matrices, exactly the same techniques you learn there will be helpful here.

8. Let us also print out the species to which these 150 flowers belong. Replace the print statement with `print iris.target`. When run, your program will now print a vector of integers: a value of zero indicates that flower belongs to the first species of Iris, a value of one that it belongs to the second species of Iris, and a value of two that it belongs to the third species of Iris.

9. Now, let's visualize this data. Replace the print statement with the following:

   (a) `plt.figure()`

   (b) `x = iris.data[:,0]`

   (c) `y = iris.data[:,1]`

   (d) `plt.scatter(x,y,c=iris.target)`

   (e) `plt.show()`

   You should get an image that looks somewhat like Fig 1b, except the colors are different. `x` stores the first feature values for all 150 flowers, `y` stores the second feature values for all 150 flowers, and the color (`c`) of each dot corresponds to the species of that flower, as stored in `iris.target`.

10. We are now going to divide this data set into two subsets: the training set, which will store the data used to train our machine learning algorithm; and the testing set, which we will use to test how accurate it is. For the moment, comment out lines 9(a-e) by placing the pound symbol at the start of each line. (This will temporarily turn off visualization.) Add

    (a) `trainX = iris.data[::2,0:2]`

just before the commented out visualization lines. This matrix contains the even-numbered rows (indicated by the '::2' before the comma) and the first two columns from the dataset. Print this variable as well as `iris.data` itself. Confirm that you have indeed sliced the matrix correctly.

11. Add this line

    (a) `trainy = iris.target[::2]`

    which captures the species of each of the 75 flowers stored on the even-numbered rows of `iris.data`. The uppercase X and the lowercase y indicate that trainX stores a matrix and trainy stores a vector. Print trainy as well as data.target, and make sure you have indeed sliced the correct data into trainy.

12. Now remove the print statements from your code and uncomment the visualization lines. Now, change lines 9(a-c) so that you draw the points corresponding to these 75 flowers, rather than the whole data set. You should now see an image similar to Fig. 1a except for different colors.

13. Now let's create the test set. Just before the visualization lines, extract the odd-numbered rows and first two columns from iris.data and store them in `testX`. **Hint:** Even-numbered rows (or columns) are referenced using `::2`; odd-numbered rows (or columns) are referenced using `1::2`.

14. Extract the odd-numbered elements from `iris.target` and store them in `testy`.

15. In your visualization lines, add a second call to `scatter` that adds the test data to the plot. You should now see an image that looks like Fig. 1b except the colors are different.

16. Add these lines

    (a) `colors = np.zeros((3,3),dtype='f')`

    (b) `colors[0,:]  = [1,0.5,0.5]`

    (c) `colors[1,:]  = [0.5,1,0.5]`

    (d) `colors[2,:]  = [0.5,0.5,1]`

    just before line 9(a). They store a light red, light green, and light blue color on the first, second, and third row of the `colors` matrix, respectively. (If you are not familiar with specifying red/green/blue colors, read this.)

17. Now we will draw each point separately, so that we can better control the color of each point.

18. Comment out the calls to `scatter`, and create a for loop that draws a single point each time through it:

    (a) `[numItems,numFeatures] = iris.data.shape`

    (b) `for i in range(0,numItems/2):`

(c) `plt.scatter(trainX[i,0],trainX[i,1])`

Line (a) captures the 'shape' (i.e., the number of rows and columns) of the data set. Line (c) draws each point by extracting the feature values from the ith row in `trainX`.

19. Let us now color each point differently, depending on the class (in this case, the Iris species) to which that point corresponds. Between lines (b) and (c) insert

    (a) `itemClass = int(trainy[i])`

    (b) `currColor = colors[itemClass,:]`

    Line (a) extracts the class of the ith item in the training set; line (b) creates a vector of red, green and blue values that corresponds to the color for that class.

20. Now add `facecolor=currColor` as an additional argument in your call to `plt.scatter` on line 18(c). You should now see a color scheme similar to that in Fig. 1a.

21. You can also make the points larger by adding `s=50` as an additional argument in your call to `scatter`. (s is short for 'size').

22. Finally, widen the edges of the points by adding `lw=2` (lw is short for 'line width'). You should now have an image identical to that of Fig. 1a.

23. Now let us create an instance of the K nearest neighbor classifier. Add

    (a) `clf = neighbors.KNeighborsClassifier(15)`

    (b) `clf.fit(trainX,trainy)`

    any where before line 9(a).

    (Also, change line 3(c) to `from sklearn import neighbors, datasets`.)

    Line (a) creates a classifier ('clf') that will predict the class of a given point. The classifier will look at the 15 points that are nearest to the new point; compute the majority class of those 15 points (i.e., which class is most represented within this set); and use that to predict the class of the new point. (In this case then, $K = 15$.) The logic of this algorithm is presented visually here. Line (b) then 'trains' the classifier using the training data. For the K nearest neighbor algorithm, this means that when a new point needs to be classified, it will use this training data to do so.

24. Let us test our trained algorithm. Immediately after line 23(b) add

    (a) `actualClass = testy[0]`

    (b) `prediction = clf.predict(testX[0,0:2])`

    (c) `print actualClass, prediction`

This will extract the class of the first item in the test set (line (a)), use your classifier to predict its class (line (b)), and print out the actual and predicted class. Are the classes the same? **Note:** On line (b), we only supply the classifier with the first two feature values of the item (i.e. the first two columns (0:2)).

25. Change lines 24(a-c) to predict and print the class of the second item in the test set. Is *that* prediction correct? Try a few more points from the test set. How well does your classifier seem to be doing?

26. Let's visualize how well it is doing. Copy and paste a second version of the loop from lines 18(b-c) just below the first loop. Now change this second loop so that it prints the points from the test set, instead of from the training set. This should produce an image almost like that in Fig. 1b, with one exception: the edge colors of the new points are different.

27. Let us enhance our visualization so that we can see what the predictions of our classifier are. To do so, we will color the *edges* of the new points based on the predicted class. So, the inner color of the new points will represent their actual class, and the edge color will represent the predicted class. This means that if the prediction is correct, both colors should be the same and the dot should all be the same color. If the dot shows different inner and outer colors, that indicates that the prediction was wrong. (Examples of both correct and incorrect predictions can be seen in Fig. 1b.) To accomplish this, within the second loop, add

    (a) `prediction = int( clf.predict( testX[i,:]  )  )`

    (b) `edgeColor = colors[prediction,:]`

    which extracts a prediction from the classifier for the ith item in the test set (line (a)), and extracts the color for that prediction from the `colors` matrix (line (b)).

28. Now add `edgecolor=edgeColor` as an additional argument in the call to `plt.scatter` in this second loop. (**Note:** Python is case sensitive, so Python knows that it should set the edge color of the circles (`edgecolor`) to the color stored in the variable `edgeColor`.) You should now have an image which is *identical* to that in Fig. 1b.

29. In order to visualize how our classifier is working, we have had to hobble it: we have only allowed it to use two of the four features that are available for this data set. If we used all four features, we would have to create a four dimensional image! Let us now quantify how well our classifier is doing so we can shut off the visualization and then use all four features. To do this, count how many predictions the classifier gets right: keep a counter that is set to zero just befor the second `for` loop, and is incremented inside the loop every time `prediction` is equal to the actual class for that item. If you print out this counter when the loop terminates, you should get a value of 60: the classifier should get 60 of the 75 test points correct.

30. Divide the counter by the total number of items in the test set, and multiply by 100. This value, if you print it out, will now report the percent of test points that the classifier classified correctly. **Note:** Python can make mistakes with division if it thinks the values should remain as integers. If you are seeing a zero value, you can place a period after numbers to indicate

that the numbers should be treated as floating point values rather than integers, or you can wrap a variable in `float()` to make Python interpret it as an floating point value. You should now get a value of 80.0, indicating that your classifier gets 80% of the test items correct.

31. Comment out every line in your code that begings with `plt`. This will shut down visualization.

32. Now go through your code from top to bottom. Wherever these is a reference to the first two columns (0:2) of either the training or test data, replace it with a reference to all four columns by replacing '0:2' with ':'. You should now see that the classification accuracy of your classifier jumps from 80% to 92%.

33. Try also changing the number of neighbors that your classifier uses by replacing the number on line 23(a) with 10 or 20. Does the classifier get better, or worse? If you use a very small number of neighbors, or numbers close to the number of items in the training set, the classifier performs very poorly. Why do you think this is so?

34. Put the number of neighbors back to 15, and now feed the classifier the second and third columns of the training and testing data: do this by putting '1:3' in the right places when you create `trainX` and `testX`. If you do this correctly, your classifier should achieve a classification accuracy of 93.333%.

35. Now uncomment all of the lines that start with `plt`. Run your code again, and it will now visualize the classification ability of your classifier using only the second and third features.

36. This week you will submit this last image, rather than a video. To do so...

    (a) Screen capture this image.

    (b) Upload it to imgur.com

    (c) Submit the URL that points to the uploaded image in BlackBoard by the deadline.