CS228: Human Computer Interaction

Deliverable 4

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 number (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., known as **kNN** for short. 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 a new Python library: matplotlib. If you cannot import matplotlib using

   ```
   import matplotlib.pyplot as plt
   ```

   that means you will have to install it. Consult the matplotlib page for instructions about how to do so for your platform.

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

   (a) `print(1)`

   Run it. Add it to your git repository.

3. You will now download two files. The first one, iris.csv, contains data about 150 iris flowers. We will use it shortly to test the kNN algorithm. Download this file, move it into the directory that contains your files, and add it to your git repository.

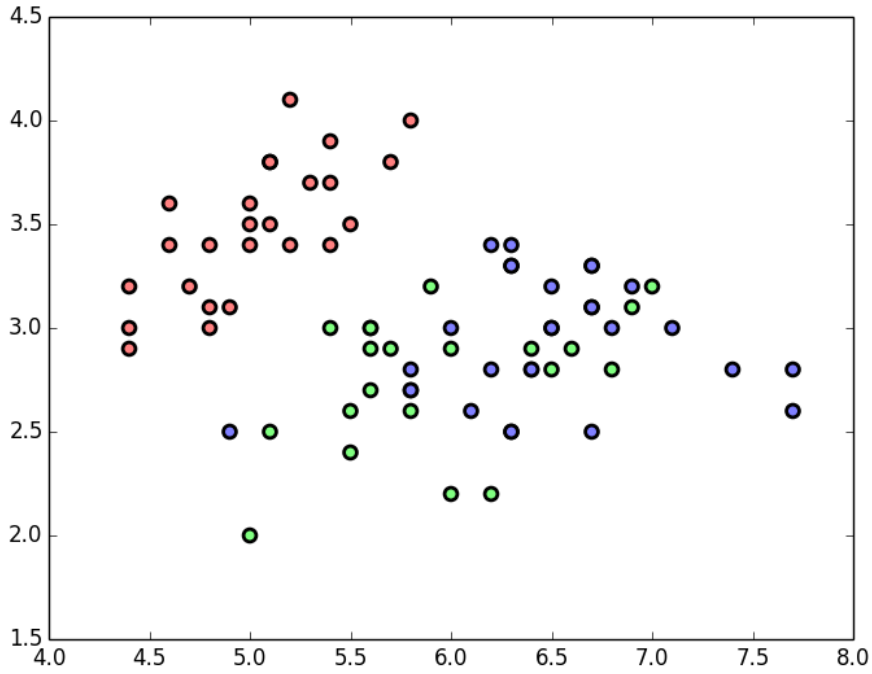
4. The second file, knn.py, contains the kNN algorithm itself. Move this file into the directory that contains your files, add it to your git repository, commit your changes, and check that your repository contains these three new files.

5. For those familiar with the `scikit-learn` library, you may be wondering why we are not making use of the kNN algorithm that comes bundled with it. This is because `scikit-learn` requires Python 3.5 or newer. Most students are using Python 2.7 because the `leap` library requires it.

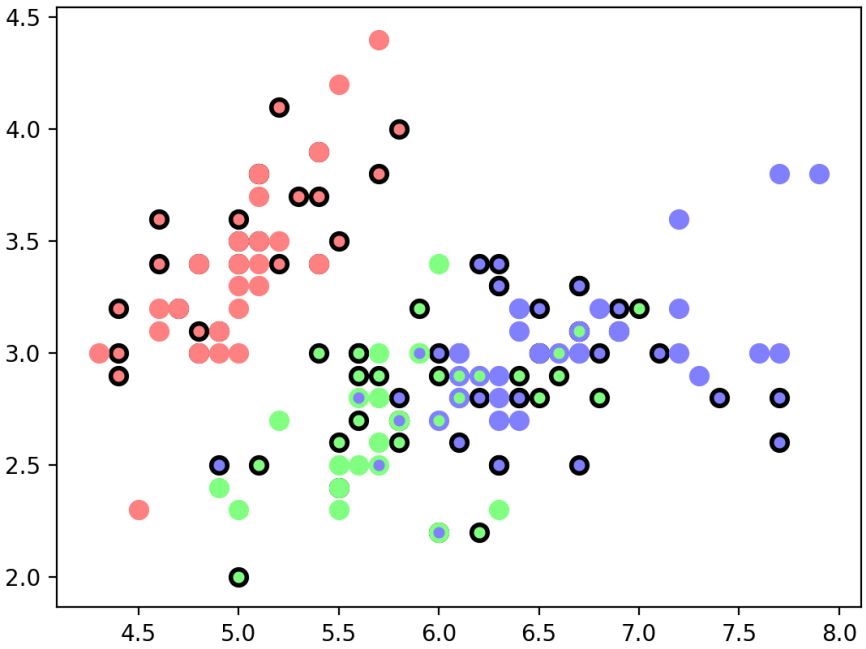
6. Inside `Predict.py`, import the `KNN` class that is found inside of `knn.py`, just like you did when you included the other classes you've made in the first three deliverables.

7. Create an instance of the `KNN` class called `knn` at the top of your code. We are now going to enable `knn` to perform some machine learning.

Importing the data set.



a



b

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

8. To start, we must provide `kNN` with some training data to work on. To do so, replace line 2a with

   (a) `knn.Load_Dataset('iris.csv')`

   This will load the iris data set, which is stored in a comma-separated values file format. You can see the data itself by opening `iris.csv`. This 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. The fifth column encodes which species that particular flower belongs to (0 = *Iris setosa*, 1 = *Iris virginica*, and 2 = *Iris versicolor*).

9. *Test your knowledge:* How many of each type of flower are recorded in the data set?

10. Please read just the introductory text about this data set on its Wikipedia page.

11. The data is stored in `knn.py` in a particular format: `data` is a 150×4 numpy matrix that stores the four features for each of the 150 flowers, and `target` is a 150-element numpy vector that stores the type of each of the 150 flowers.

12. To see this, add `print(knn.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). You might see that the numbers seem to have round-off errors: for example, a value stored as 6.7 in the csv file may be printed as 6.6999981. This is an artefact of numpy that we can safely ignore.

13. Change this last line to `print(knn.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.

14. **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.

15. Let us also print out the species to which these 150 flowers belong. Replace the print statement with `print(knn.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.

Visualizing the data set.

16. Now, let's visualize this data using the matplotlib library. Make sure you have this line

```
import matplotlib.pyplot as plt
```

at the top of your program. Matplotlib is a library that contains several sub-libraries; `pyplot` is one of them. We are creating a shorthand here, `plt`, so we do not have to write out `matplotlib.pyplot` every time we want to access functions inside this sub-library.

17. Replace the print statement with the following:

    (a) `plt.figure()`

    (b) `plt.show()`

    When you run your code now, you should see an empty matplotlib window. Matplotlib is similar to pygame in that it creates a new figure window and then shows it to the user. We are using matplotlib instead of pygame here because it is much easier to plot data in matplotlib than it is in pygame.

18. We now want to capture the first features of all 150 flowers and store it in a variable called x. These values are stored in the first column of `knn.data`. Use numpy slicing to extract this data and store it in a variable called x. Print this variable. How many values are printed? How many did you expect to be printed? Are you grabbing the correct values? Open `iris.csv` to check.

19. Now capture the second feature of all 150 flowers, again using slicing, and store it in another variable called y. Print it. How many values are printed? How many did you expect to be printed? Are you grabbing the correct values?

20. Now let us draw the data using a scatter plot by putting this line

    (a) `plt.scatter(x,y,c=knn.target)`

    between `figure()` and `show()`.

    You should get an image that looks somewhat like Fig 1b, except the colors and circle sizes may be 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 `knn.target`.

21. 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 the model produced by our algorithm is. For the moment, comment out all the `plt`-related lines by placing the pound symbol at the start of each line. This will temporarily turn off visualization.

22. Add

    (a) `trainX = knn.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. Consult `iris.csv` to confirm that you have indeed sliced the matrix correctly.

23. Add this line

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

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

24. Now remove the print statements from your code and uncomment the visualization lines. Now, change line 20a so that you draw the points corresponding to these 75 flowers, rather than the whole data set.

    Hint: The first column of `trainX` is now your x data. Where should the y data come from?

    Hint: The colors should be drawn from `trainy` rather than from `knn.target`.

    You should now see an image similar to Fig. 1a except for different colors.

25. Now let's create the test set. Just before the visualization lines, extract the odd-numbered rows and first two columns from `knn.data` and store them in `testX`. Print the captured data and consult `iris.csv` to check your work.

26. Extract the odd-numbered elements from `knn.target` and store them in `testy`. Print the captured data and use `iris.csv` to check your work.

27. 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, and the ranges of the axes may also be different.

28. 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 the visualization lines. 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.)

    Line (a) may require you to import `numpy` to this file using `np` as shorthand, in the same way we did for matplotlib.pyplot on line 16.

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

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

    (a) `[numItems,numFeatures] = knn.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`.

Check your understanding: Why are we iterating over only half of the number of items?

31. You will notice that each circle is drawn with the same color. Do you know why? This web page may give you a clue.

32. 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) above 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.

33. Now add `facecolor=currColor` as an additional argument in your call to `plt.scatter` on line 30(c). You should now see a color scheme similar to that in Fig. 1b.

34. 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').

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

Applying kNN to the data set.

36. Now let us tell `knn` how to use the data we give it (the training data) to make predictions about data it has not yet seen (the test data).

    To start, add

    (a) `knn.Use_K_Of(15)`

    (b) `knn.Fit(trainX,trainy)`

    anywhere before the visualization lines but after `trainX` and `trainy` have been created.

    Line (a) tells `knn` how to make predictions. In this case, it 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. The logic of this algorithm is presented visually here.

    Line (b) then tells `knn` what data to use for making predictions about new data points it has not seen before.

37. Let us test our trained algorithm. Immediately after the call to `Fit(...)` add

(a) `actualClass = testy[0]`

(b) `prediction = knn.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)).

38. Change the lines in step 37 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?

39. Create a loop that iterates over all 75 elements in the test set. How many did `knn` get right?

Visualizing kNN's performance.

40. Let us visualize how well it is doing. Copy and paste a second version of the loop from step 30 just below the first loop. Now change this second loop so that it draws the points from the test set, instead of from the training set. This should produce an image almost like that in Fig. 1b: you are now drawing the training and test data. One difference should be apparent however: the edge colors of the new points are different. (Do not worry if the ranges of the axes are different.)

41. 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( knn.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)).

42. 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.

43. 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 before 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.

44. 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. What value do you expect to see? What value do you see?

    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.

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

46. 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%.

47. Try also changing the number of neighbors that your classifier uses by replacing the argument to `Use_K_Of(...)`. 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?

    Note: for this relatively simple data set, small values of k may actually work well. However, this is generally not true for more complex data sets.

48. 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%.

49. Now uncomment all of the lines that draw and display the training and test data. Run your code again. It will now visualize the classification ability of your classifier using only the second and third features. It should look similar to, but not identical to Fig. 1b. Do you understand why?

50. 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.