CS228: Human Computer Interaction

Deliverable 5

Important: Please replace any references to `numpy.save()` and `numpy.load()` with `pickle.dump()` and `pickle.load()` in the programs you use/create for this deliverable. (This will require you to add 'import pickle' at the top of all of your programs.) At the end of this deliverable, you will be uploading data files that become part of a class-wide data set: if everyone uses pickle, all of the files will be in the same format.

Description

In the previous deliverable, you created a k Nearest Neighbor (kNN) classifier that learned to predict which of three species of Iris a flower was, if given four features that were measured from that flower.

In this deliverable, you will adapt your classifier to work with Leap Motion data: the classifier will take as input hand data from a single frame, and indicate which of two ASL numbers the classifier thinks the user signed. In this deliverable we will be working with the ASL numbers from zero to nine; these signs are shown in Fig. 1. Each student will be working with two of these digits. To see which digits you are assigned, refer to Table 1.

Instructions

1. First, we will modify the `Record.py` program you made in Deliverable 3 to save out four data files: a training and testing set for each of the two numbers you are assigned. Start by creating a new directory (`Deliverable5`) and copying and pasting your `Record.py` into it.

2. Recall that `Record.py` saves one frame of data from your primary hand at the moment that your secondary hand leaves Leap Motion's field of view. In other words, when the red hand (indicating 'record') turns green (indicating 'no record'). (Video overview here.) We are going to modify `Record.py` so that now, it records *every* frame from the primary hand while the secondary hand is in view. This collection of gestures will then become the training (or testing) set for your first (or second) assigned number. Let's start by adding a new variable, `numberOfGestures`, to the `Deliverable` class in `Record.py`, and assign it a value of 100: we will start by recording 100 gestures at a time. Run your program; there should be no change in the program's behavior.

3. Now add a second variable, `gestureIndex`, to the class (and set it to zero). This variable will keep track of how many gestures have been recorded.

4. We are now going to expand the `gestureData` variable from a three-dimensional matrix to a four-dimensional matrix. To do so, change the definition of `gestureData` to:

   (a) `self.gestureData = np.zeros((5,4,6,self.numberOfGestures),dtype='f')`

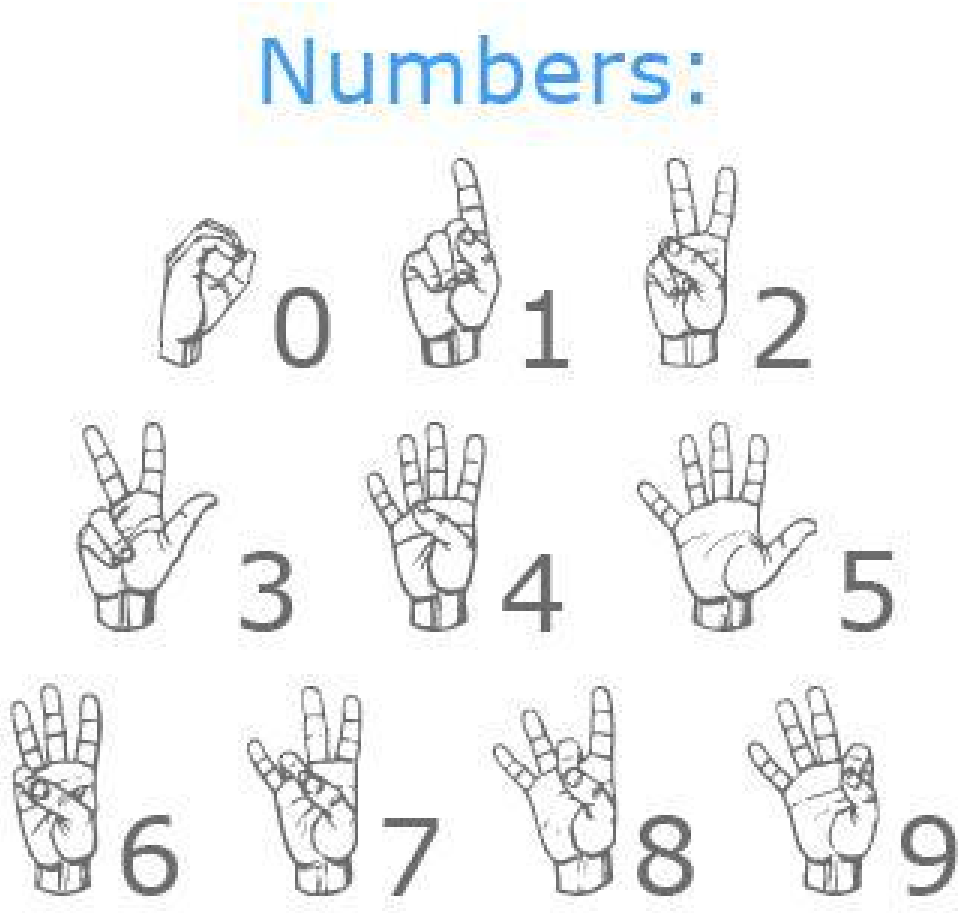

Figure 1: The first 10 ASL numbers

This has now made room in this variable to store 100 gestures.

5. Now locate the lines in `HandleBone` where the coordinates of the bone's base and tip are stored in `gestureData`. Place all six of those lines inside an `if` statement and modify them as follows:

```
(a) if ( self.currentNumberOfHands == 2 ):
(b)       self.gestureData[i,j,0,self.gestureIndex] = xBase
(c)       self.gestureData[i,j,1,self.gestureIndex] = yBase
(d)       ...
```

Since `gestureIndex` is equal to zero, this will store every captured frame of the primary hand in the first 'frame' of gestureData. That is, in the 3D matrix `gestureData[:,:,:,0]`. To ensure that this is so, add `print gestureData[:,:,:,0]` after the sixth line inside this `if` statement. When you run your program now and record data, all the numbers inside this matrix should be non-zero.

6. To ensure that the next frame is empty (and ready to stored a recorded gesture, change the print statement to `print gestureData[:,:,:,1]`. Now when you run your program, all the values should be zero.

7. (Delete the print statement.) Now, instead of continuously overwriting `gestureData[:,:,:,0]`, let's store 100 gestures in `gestureData`. Do so by placing the following lines at the bottom of the function `HandleHands`:

   (a) `if ( self.currentNumberOfHands == 2 ):`

   (b) `    print 'gesture ' + str(self.gestureIndex) + ' stored.'`

   (c) `    self.gestureIndex = self.gestureIndex + 1`

   (d) `    if ( self.gestureIndex == self.numberOfGestures ):`

   (e) `        sys.exit(0)`

   This will ensure that when the current frame has been stored, `gestureIndex` is incremented such that during the next pass through `HandleHands`, the new frame will be stored in another part of `gestureData`. Also, if 100 gestures have been stored (d), the program will exit (e). Run your program now. You should see it print messages whenever you are 'recording'. The program should now quit on its own.

8. Between lines 7(d) and 7(e), put a few `print self.gestureData[:,:,:,k]` statements. Try $k = 0$ and $k = 99$ to ensure that both the first and final 'frame' of `gestureData` is filled with numbers before the program quits.

9. Now let's save this data that you've collected before the program quits. **(Note: make sure to use pickle.dump rather than numpy.save.)** Within the `HandleHands` function, delete the lines that call `SaveGesture`. Now place a call to `SaveGesture` just before line 7(e). Modify `SaveGesture` so that it saves `gestureData` to the file `userData/gesture.dat`. Delete the lines in `SaveGesture` that save out the second file `userData/numOfGestures.dat`. Finally, get rid of the variable `numberOfGesturesSaved` as we don't need it anymore. Run your program until it saves a data set to `userData/gesture.dat` and quits.

10. Let's make sure that the data was saved correctly. Create a new empty Python program called `Classify.py`. Import `numpy` and `pickle` at the top as you do in `Record.py` and load `userData/gesture.dat` into a variable called `gestureData` (**using pickle.load rather than numpy.load**). Print this variable and ensure that the subset of numbers in the variable that are shown are non-zero.

11. Let's check the 'shape' of this variable. Change the print statement to `print gestureData.shape`. When you run your program now it will show you four numbers: the number of rows in the matrix (i.e. the length of the first dimension); the number of columns (i.e. the length of the second dimension; the length of the third dimension; and the length of the fourth dimension. You should recognize these numbers, and recall which part of the data set is stored along which dimension.

12. Before we can expand `Classify.py` to perform classification, we need to record some data sets for it to work on. Go back to `Record.py` and change the number of gestures to be saved from 100 to 1000.

13. Now, practice signing your first assigned number while `Record.py` is running (but before you trigger recording using your secondary hand). You need the Leap Motion device to capture the number correctly, but you will also need to twist your hand and wrist a bit to make sure that the number is captured at different orientations (don't worry about different positions for now). This video will give you an idea how to practice.

14. When you're happy that the number is being captured faithfully by Leap Motion, bring your secondary hand in and record 1000 gestures, while gently twisting your hand and wrist. When the program stops, rename `gesture.dat` to `trainM.dat`, where M should be your number. (For example if your first assigned number is 3, change the file name to `train3.dat`).

15. Now make a second recording, this time of your second number, and rename the resulting `gesture.dat` to `trainN.dat`, where N is equal to your second number.

16. Perform a third recording with your first number and rename the resulting `gesture.dat` to `testM.dat`.

17. Perform a fourth and final recording with your second number and rename the resulting `gesture.dat` to `testN.dat`. You should now have four data files in your `userData` directory.

18. (You can shut off your Leap Motion device now for a while.) Return to `Classify.dat`, and load these four files into the variables `trainM`, `trainN`, `testM`, and `testN`. Print all four matrices to make sure all the shown numbers are non-zero, and print their shapes. Each matrix should have the same shape (and the same shape as `gestureData` when your printed its shape during step 11). Remove `gestureData` from the program; you don't need it anymore.

19. In order to supply this data to the kNN classifier, we need to reshape these matrices into a form that the classifier can digest. Recall that the classifier requires a matrix X and a vector y: each row in X corresponds to a training point, and each column corresponds to a feature that describes those points. Each element in y stores an integer value, which indicates which class that point belongs to. In our case, we have now have $1000 \times 2 = 2000$ training points, and each point is described by $5 \times 4 \times 6 = 120$ features. So let's start by creating a function called `ReshapeData` as follows:

```
(a) def ReshapeData(set1,set2):
(b)     X = np.zeros((2000,5*4*6),dtype='f')
(c)     for i in range(0,1000):
(d)         n = 0
(e)         for j in range(0,5):
```

```
(f)                 for k in range(0,4):
(g)                     for m in range(0,6):
(h)                         X[i,n] = set1[j,k,m,i]
(i)                         n = n + 1
(j)       return X
(k) trainX = ReshapeData(trainM,trainN)
(l) print trainX
(m) print trainX.shape
```

Take a moment to read and understand this function before incorporating it into `Classify.py`. Note how the function marches down the first 1000 rows in `X` (lines (c) and (h)). At each row, it then marches from left to right using n as an index (lines (d) and (i)). It fills in each element $[i, n]$ with a position coordinate from the ith gesture stored in `set1`. If you run `Classify.py` now, you should see that the first 1000 rows are populated with values, but the lower 1000 rows are still all zeros.

20. Let's fill in those bottom 1000 rows using the gesture data stored in `set2`. To do so, add a line between lines 19(h) and 19(i) that stores each element $[i+1000, n]$ in `X` with a coordinate from the ith gesture stored in `set2`. In other words, fill the top 1000 rows with gestures of your first assigned number, and the bottom 1000 rows with gestures from your second assigned number. When you run your code now you should see that all the shown numbers from `trainX` are non-zero.

21. Inside `ReshapeData`, let us now create and return our y vector containing the class labels. After line 19(b), create a vector y with length 2000.

22. After line 19(h), store the value of your first assigned digit in `y[i]`. (For example, if my First Number in Table 1 is '8', I would store the number 8 in `y[i]`.) This indicates that the first 1000 points in `X` belong to class '8' (for example).

23. After *that* line, store the value of your second assigned digitin `y[i+1000]`. (For example, if my Second Number in Table 1 is '9', I would store the number 9 in `y[i+1000]`.) This indicates that the second 1000 points in `X` belong to class '9' (for example).

24. Change line 19(j) to return both `X` and `y`.

25. Change line 19(k) to receive two variables from `ReshapeData`: `trainX` and `trainy`.

26. Finally, print `trainy` and its shape. You should see that the first 1000 values are your first number, and the second thousand values are your second number.

27. Now let's create the test data. Add a new line after 19(k) that calls `ReshapeData` again, but this time with `testM` and `testN` and stores the resulting `X` and `y` in in `testX` and `testy`. Print these variables' contents and shapes to make sure they are filled correctly.

28. We're now ready to do some classification. Include `sklearn` exactly as you did in the last deliverable at the top of `Classify.py`.

29. After the lines that reshape your training and test data, create a kNN classifier

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

    and train it:

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

30. Now let's see how good your classifier is. After the above line, create a loop that iterates through the 2000 test points.

31. Within the loop, supply the ith point stored in the ith row of `testX` to the classifier and capture the resulting class prediction in a variable called `prediction`.

32. Compare the prediction against the actual class stored in `testy[i]`. Increment a counter if the prediction was correct.

33. Print the counter when the loop finishes. How many of the 2000 test points did your classifier get right?

34. Convert `numCorrect` to a percentage. If the prediction success rate is near 50%, something is wrong: your classifier is not doing better than just flipping a coin and predicting the class that way. If your prediction rate is near 50%, go back and check your code to see if there any errors you missed. If you can't find any errors, try going back and recording your training and test data again: perhaps Leap Motion failed to capture one or more of the data sets correctly. Once you have a success rate significantly above 50%, continue to the next step. **Guide**: I used the digits zero and one, and got a percent rate of about 91%. However, for digits that are somewhat similar to one another, your success rate may not be this high.

35. Let's now try to 'clean' our training and test data so that we can increase the classifier's performance. The first thing we'll do is remove some of the 120 features so that the data lives in a lower dimensional space.

36. In `Classify.py`, Create an empty function called `ReduceData(X)` (you can do this by including the single line `return X` inside it). Then, call this function just after each of the four data sets have been read in from their respective files, but before they've been reshaped:

    (a) `trainM = ReduceData(trainM)`

    (b) `trainN = ReduceData(trainN)`

    (c) `testM = ReduceData(testM)`

    (d) `testN = ReduceData(testN)`

    Re-run your code to make sure you've returned each data set to itself: if you have done it correctly, you should not see a change in the prediction accuracy of your classifier.

37. Now let's cut out some of the data that is redundant. Look at your hand for a moment. Extend and flex your index finger while rotating your hand. Think about the position of the base of your index finger and the tip: as long as you know what those two positions are, you also know the position of the two joints 'inside' your index finger. In other words, you can't rotate those two inner joints without also rotating the position of your finger's tip. So: we can throw away information relating to the intermediate and proximal phalanges in each finger (consult Fig. 2). These correspond to the middle two columns (all the way back) of the 3D matrix: that is, `gestureData[:,1,:,:]` and `gestureData[:,2,:,:]`. So how do we delete them? We do so by including these two lines

    (a) `X = np.delete(X,1,1)`

    (b) `X = np.delete(X,1,1)`

    in `ReduceData`. The first call cuts out the second instance along the second dimension of `X`: that is, the second column (i.e. the proximal phalange). The smaller `X` now has three columns. The second call cuts out the second column again, which now corresponds to the intermediate phalange. Before we can run our code however, we have to change line 19(f) to reflect the fact that we are now iterating over two bones instead of four. We also need to change the shape of `X`, as defined on line 19(b). Run your code; did the prediction accuracy increase? Perhaps a bit.

38. What other data can we cut out? If you think about it, we probably don't need the base positions of the remaining bones, but just their tips. The base coordinates correspond to `gestureData[:,:,0:3,:]` (consult Fig. 2c). So, delete the first instance along the third dimension of `X` in `ReduceData`, three times (`X = np.delete(X,0,2)`). Note also that you need to change another number in the nested loop shown in lines 19(a-m): which number is it, and what should it be changed to? Also remember to change the shape of `X`.

39. Let's do one more bit of cleaning: let's center each gesture. Imagine that you generate two training points: you sign '0' to the left of Leap Motion's field of view, and you sign '1' to its right. Now, another user comes along and signs '1' in its left field of view. This single test point will probably be closer to the '0' training point than to the '1' training point, so the classifier will probably produce an incorrect prediction. Imagine however that all gestures are moved so that they are centered around the origin (0,0,0). This problem can then not happen. Let's start by defining a new function called `CenterData(X)`, have it return `X` to start, and call it on each of the four data sets immediately after you call `ReduceData(X)` on them. Make sure that your classifier's prediction accuracy doesn't change.

40. Inside `CenterData`, grab all of the x coordinates

    `allXCoordinates = X[:,:,0,:]` (refer to Fig. 2a),

    compute the mean of all those coordinates

    `meanValue = allXCoordinates.mean()`,

    and then offset each of those coordinates by their mean value

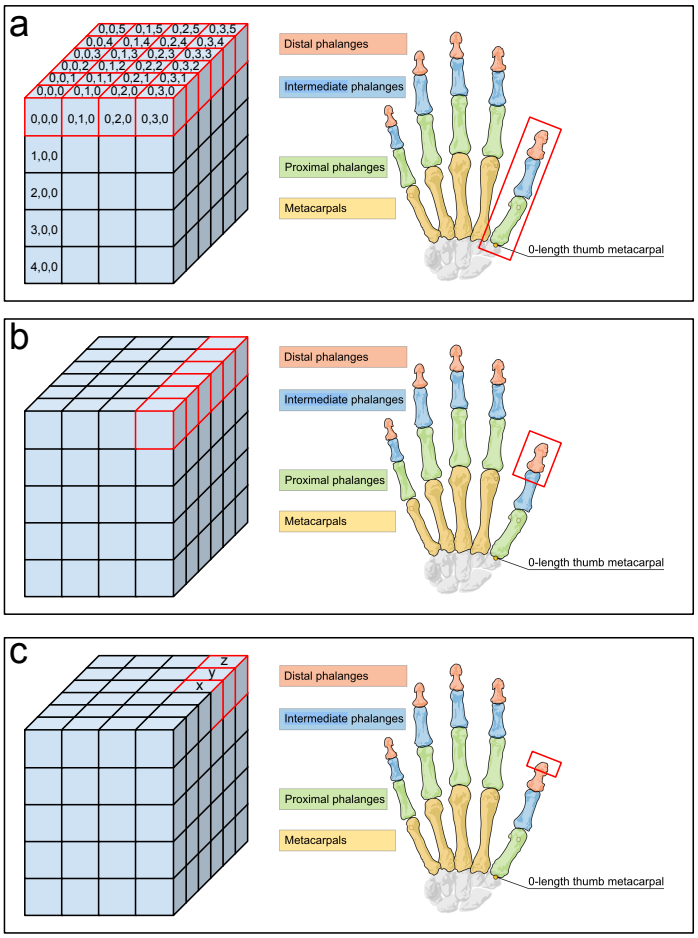

Figure 2: The stored gesture data and the corresponding hand anatomy (from Deliverable 3).

```
X[:,:,0,:]  = allXCoordinates - meanValue
```

just before returning `X`.

41. To check that you indeed centered the gesture along the x axis, print `X[:,:,0,:].mean()` just before returning `X`. What value do you expect to see? Do you see this value? (Hint: the expected number might be just a bit different than the one you are expecting because of rounding off errors during these calculations; no need to worry about this.) Does your classifier's prediction accuracy increase? (You can remove the print statement now.)

42. Copy and paste the three lines in step 40, and modify them to center the y coordinates in each of the four data sets as well. Run your code again, and print out the mean value of the centered y coordinates. Do they match the value you were expecting? (Remove the print statement.) Did your prediction accuracy increase?

43. Finally, copy and paste again, and modify the new lines to center the z coordinates. Check that they were correctly centered, and check the prediction accuracy: did it increase?

44. Comment out (but don't delete!) the calls to `ReduceData` and `CenterData`. Run your code again, and write down the resulting prediction accuracy (as a percentage).

45. Uncomment the calls to ReduceData, and write down the new prediction accuracy.

46. Uncomment the calls to CenterData, and write down this third prediction accuracy.

47. In your submission to BlackBoard for this deliverable, please type in five numbers directly to your submission: your two assigned numbers, followed by these three prediction accuracies. Finally, attach your four data files (`trainM.dat`, `trainN.dat`, `testM.dat`, and `testN.dat`) to your submission and submit.

Table 1: Assigned numbers

Name	First Number	Second Number
Boisjoli, Brian T.	0	1
Brooks, Mitch A.	1	2
Bryant, David	2	3
Casey, Steven J.	3	4
Chen, Ziye	4	5
Driscoll, Andrew F.	5	6
Dryzga, Kevin C.	6	7
Dzwonar, Emilie R.	7	8
Gaspero-Beckstrom, Fabian S.	8	9
Girdzis, Gretchen E.	9	0
Glade, Max R.	0	1
Gottfried, Kevin N.	1	2
Gray, Kelly	2	3
Hewgill, Blake W.	3	4
Jones, Andrew R.	4	5
Kazour, Michael J.	5	6
Kruger, Kolby R.	6	7
Libby, Preston E.	7	8
Lovjer, Ivan	8	9
Lynch, Michael K.	9	0
Marks, David	0	1
Maynard, Kai	1	2
McCracken, James M.	2	3
Pakulski, Joe F.	3	4
Palmer, Madison A.	4	5
Palmer, Shawn O.	5	6
Pornelos, Jon P.	6	7
Quisenberry, Scott K.	7	8
Rice, William M.	8	9
Sandvik, Chris J.	9	0
Siebert, Joseph M.	0	1
Steffens, Jack A.	1	2
Swanke, Samuel	2	3
Turner, Collin D.	3	4
Vickroy, Cody W.	4	5
Xiao, Yao	5	6
Zacchea, Abigail R.	6	7