PIC 16: Homework 6 (due 5/19 at 10pm)

How should your answers be submitted?

- Download `hw6.py`. Replace `mjandr` by your name.
- Give a better definition of `preprocess`, and submit `hw6.py` to CCLE.

In this assignment, you will preprocess poker hand data to help an SVC learn poker hands.

1. In the zip file which you downloaded, you'll find `hw6.pdf` (this file), `hw6.py`, `hw6_checker1.py`, `hw6_checker2.py`, and `trick.json`.

   - You will edit and submit `hw6.py`.
   - You can also edit `hw6_checker1.py`, `hw6_checker2.py` as much as you like. BUT keep in mind that the grader will be running `hw6_checker1.py`, `hw6_checker2.py` UNEDITED.

2. Both files `hw6_checker1.py` and `hw6_checker2.py` create a list called `hands` whose elements are numpy arrays. Each array has `ndim == 2` and `shape[1] == 5`: they store hands consisting of `5` cards.

   A deck of cards has 52 cards. The convention I have used is that...

   - `0 1 2 ... 10 11 12` correspond to `A 2 3 ... 10 J Q K` of the first suit;
   - `13 14 15 ... 24 25` correspond to `A 2 3 ... 10 J Q K` of the second suit;
   - `26 27 28 ... 37 38` correspond to `A 2 3 ... 10 J Q K` of the third suit;
   - `39 40 41 ... 50 51` correspond to `A 2 3 ... 10 J Q K` of the fourth suit.

   So

   - `np.array([2, 13, 26, 28, 41])` encodes `3 A A 3 3` of various suits (a full house).
   - `np.array([1, 14, 20, 27, 40])` encodes `2 2 8 2 2` of various suits (four of a kind).

3. `hw6_checker1.py` attempts to use an SVC to learn the difference between full houses, four of a kinds, and straight flushes.

   `hw6_checker2.py` attempts to use an SVC to learn the difference between one pairs, two pairs, three of a kinds, straights, flushes, full houses, and four of a kinds.

   Your first objective is to understand the code I have written. You do not need to worry about how I generated `hands`. If you really want to know, I am happy to show you. But you do need to understand line 45 and onwards. See over the page for more help.

4. You'll find both bits of code do quite badly (about 70% and 60% at best).

   Edit the `preprocess` function in `hw6.py` so that they do better.

   My `hw6_checker1.py` always gets 100%.

   `hw6_checker2.py` is more likely to struggle. Mine hits 100% about 9 out of 10 times, but with unlucky training and testing sets can be as bad as 99.29%.

Some comments about my code.

1. In `hw6_checker1.py`,

   (a) `hands[0]` has shape `(3744,5)` and consists of all full houses;
   (b) `hands[1]` has shape `(624,5)` and consists of all four of a kinds;
   (c) `hands[2]` has shape `(40,5)` and consists of all straight flushes.

   In `hw6_checker2.py`,

   (a) `hands[0]` has shape `(1098240,5)` and consists of all one pairs;
   (b) `hands[1]` has shape `(123552,5)` and consists of all two pairs;
   (c) `hands[2]` has shape `(54912,5)` and consists of all three of a kinds;
   (d) `hands[3]` has shape `(10200,5)` and consists of all straights;
   (e) `hands[4]` has shape `(5108,5)` and consists of all flushes;
   (f) `hands[5]` has shape `(3744,5)` and consists of all full houses;
   (g) `hands[6]` has shape `(624,5)` and consists of all four of a kinds.

2. In `hw6_checker1.py`,

   (a) we train on 28 full houses and test on 50 full houses;
   (b) we train on 28 four of a kinds and test on 50 four of a kinds;
   (c) we train on 8 straight flushes and test on 20 straight flushes.

   In `hw6_checker2.py`,

   (a) we train on 100 one pairs and test on 100 one pairs;
   (b) we train on 50 two pairs and test on 100 two pairs;
   (c) we train on 50 three of a kinds and test on 100 three of a kinds;
   (d) we train on 100 straights and test on 800 straights;
   (e) we train on 100 flushes and test on 800 flushes;
   (f) we train on 50 full houses and test on 100 full houses;
   (g) we train on 50 four of a kinds and test on 100 four of a kinds.

3. The purpose of `partition` is to start with a numpy array consisting of a certain type of hand, randomize the ordering of the hands, take the first `train` and last `test` hands and return them as a tuple. The `assert` statement makes sure we have some nonzero amount of training and testing data, and that we do not test on training data.

4. The code after `partition` uses list comprehension multiple times to efficiently (in terms of the amount of code written) generate the training and testing samples, as well as the target values. If you're confused by what values these arrays contain, you should print elements until you are not confused any more.

5. Finally, we actually train the SVC and test its accuracy.

   Notice the commented out code; it might be helpful for learning about what samples the SVC is struggling with.