Programming for Probability

Authors:

v1.0 (Spring 2020) William Gan, Aditya Sengupta, Christina Zhang, Sasha Khazatsky, Kannan Ramchandran

Introduction

What's the probability that a 5 card hand in poker is a flush? What's the expected number of times you have to flip a coin until you see three heads in a row? What's the optimal way to play blackjack at a casino?

In EECS 126, we teach you techniques to answer these questions with just pencil and paper. However, in the real world, it can be useful to simply use computers to calculate these answers. In this lab, we give an overview of various techniques to do so and give you some practice.

```
In [1]: import itertools

        import numpy as np
```

Q1: Brute Force / Exhaustive Search

Sometimes it's possible to iterate through every outcome. For example, to find the probability that a 5 card hand in poker is a flush, maybe we can just iterate over every possible 5 card hand and then see how many are flushes. Since every 5 card hand is equally likely, the probability can be found via simple division.

Question: How many 5 card hands are there (exact number)?

$$\binom{52}{5} = 2598960$$

Fill in the following functions that are used to calculate if a hand is a flush. A 5-card hand is a flush if all the suits are the same and it is not a straight. A straight consists of 5 consecutive ranks e.g. 8 9 10 J Q. Ranks are not allowed to wrap around (e.g. K A 2 3 4) is not a straight; however, the ace **is** allowed to be at the front or back. So 10 J Q K A and A 2 3 4 5 are straights. You can run `test_is_consecutive` to make sure it works.

```
In [2]: RANKS = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K', 'A']
        SUITS = ['C', 'D', 'H', 'S']
```

In [3]:
```
def is_same_suit(hand):
    ### BEGIN YOUR SOLUTION
    suit = hand[0][1]
    for card in hand:
        if not card[1] == suit:
            return False
    return True
    ### END YOUR SOLUTION

def is_consecutive(hand):
    ### BEGIN YOUR SOLUTION
    assert len(hand) == 5
    numerical_ranks = [RANKS.index(card[0])+2 for card in hand]
    sorted_ranks = sorted(numerical_ranks)
    if sorted_ranks == [2, 3, 4, 5, 14]:
        return True
    for i in range(4):
        if not sorted_ranks[i+1] == sorted_ranks[i]+1:
            return False
    return True
    ### END YOUR SOLUTION

def is_flush(hand):
    assert len(hand) == 5
    return is_same_suit(hand) and not is_consecutive(hand)
```

In [4]:
```
def test_is_consecutive():
    assert is_consecutive(((8, 'C'), (9, 'D'), (10, 'H'), ('J', 'S'),
('Q', 'C')))
    assert not is_consecutive(((8, 'C'), (9, 'D'), (2, 'H'), ('J',
'S'), ('Q', 'C')))
    assert is_consecutive(((10, 'C'), ('J', 'C'), ('Q', 'C'), ('K',
'C'), ('A', 'C')))
    assert is_consecutive((('A', 'H'), (2, 'H'), (3, 'H'), (4, 'D'),
(5, 'C')))
    assert not is_consecutive((('K', 'H'), ('A', 'H'), (2, 'H'), (3,
'D'), (4, 'C')))
    print('All tests passed')
```

In [5]:
```
test_is_consecutive()
```

```
All tests passed
```

`itertools` is a built-in Python library that provides Python generators for things like permutations and combinations. For example, `itertools.product(RANKS, SUITS)` returns the tuples (2, 'C'), (2, 'D'), ..., ('A', 'S'). **Use it to iterate through every 5 card hand and calculate the probability of a flush.**

Hint: Look through the Python documentation to find `itertools.combinations`.

In [6]:
```
deck = itertools.product(RANKS, SUITS)
total = 0
flushes = 0

### BEGIN YOUR SOLUTION
for hand in itertools.combinations(deck, r=5):
    if is_flush(hand):
        flushes += 1
    total += 1
### END YOUR SOLUTION

print(flushes / total)
```

```
0.0019654015452334 78
```

Q2: Monte Carlo Simulation

Sometimes there are too many outcomes to iterate over. While a single 5-card hand is relatively simple, a multi-player poker game can have billions of outcomes. Even if you were to group some outcomes based on symmetry, there would still be too many. Some problems, like finding how long it takes to see three heads in a row, have an infinite number of outcomes.

In these situations, we can still find approximate answers via sampling. For example, to find how long it takes to see three heads in a row, maybe we can just play it out many times and take the average.

To run this simulation, we need to be able to generate random events. Python's built-in `random` module allows you to do this, but usually we want to use `np.random`. They have more or less the same functionality, but using NumPy will sometimes have a performance advantage.

In the following cell, use Monte Carlo sampling to calculate how many flips are needed to see three heads in a row.

In [7]:
```
TRIALS = 100000
running_sum = 0
for _ in range(TRIALS):
    ### BEGIN YOUR SOLUTION
    num_heads_in_a_row = 0
    num_flips = 0
    while True:
        if np.random.randint(2) == 0:
            num_heads_in_a_row = 0
        else:
            num_heads_in_a_row += 1
        num_flips += 1
        if num_heads_in_a_row == 3:
            break
    running_sum += num_flips
    ### END YOUR SOLUTION
print(running_sum / TRIALS)
```

```
13.95666
```

Q3: Birthday Paradox

How many people do you think there need to be in a room for two of them to have the same birthday with probability at least 50%? A lot of people say $365/2 \approx 180$, but it turns out the answer is a lot less. We can guess what this answer is using the following strategy.

The more people there are in the room, the higher the probablity two of them have the same birthday. In other words, the probability is monotonically increasing, and we can binary search on the number of people in the room. For a specific number of people, we can estimate the probability that two have them the same birthday via simulation.

In the following cell, use Monte Carlo sampling to determine the probability that two people have the same birthday. You can use `test_same_birthday` to verify the probabilities.

In [8]:
```
def p_same_birthday(num_people, trials=10000):
    same_birthday_count = 0
    ### BEGIN YOUR SOLUTION
    for _ in enumerate(range(trials)):
        birthdays = np.random.randint(365, size=num_people)
        if len(np.unique(birthdays)) < num_people:
            same_birthday_count += 1
    ### END YOUR SOLUTION
    return same_birthday_count / trials
```

In [9]:
```
def test_p_same_birthday():
    assert np.abs(p_same_birthday(10) - 0.11694817771107768) < 0.02
    assert np.abs(p_same_birthday(40) - 0.891231809817949) < 0.02
    print('All tests passed')
```

In [10]:
```
test_p_same_birthday()
```

```
All tests passed
```

In the following cell, use binary search to find the first value of `num_people` such that the estimated probability is at least 0.5.

In [11]:
```
def binary_search():
    ### BEGIN YOUR SOLUTION
    lo = 2
    hi = 365
    while lo < hi:
        mid = (lo + hi) // 2
        if p_same_birthday(mid) >= 0.5:
            hi = mid
        else:
            lo = mid+1
    return lo
    ### END YOUR SOLUTION
```

In [12]:
```
binary_search()
```

Out[12]:
```
23
```

Q4: Tricks

The message so far has basically been that for simple problems, we can just use brute force, and for harder problems, we can perform sampling. This is mostly true, but there is often an art that comes to both techniques. In many cases, you can exploit symmetries or be clever in other ways. We already did this with the 5-card flush probability: a more naive way would have been to iterate over all $52 \cdot 51 \cdot 50 \cdot 49 \cdot 48$ ways 5-cards could be dealt. Most "tricks" aren't as obvious, but when found, they can greatly speed up your program.

Say we wanted to find the probability a 12-card hand is "nice". A hand is nice if it can be arranged in a way that is symmetrical. For 5-card hands, A K K 5 A is nice since it can be arranged into A K 5 K A, whereas 10 10 J Q 10 isn't nice. To find this probability, we could take a similar approach to the flush program, but that would involve going through 206379406870 hands. This isn't feasible, but it turns out some hands are the same, since in this problem, the suit doesn't matter. A hand is really just a tuple with the counts of each rank. For example 10 10 J Q 10 is

2	3	4	5	6	7	8	9	10	J	Q	K	A
0	0	0	0	0	0	0	0	3	1	1	0	0

And the number of unique tuples is far lower. So what we need to do is iterate through all the unique tuples. However, we also need to explicitly find the probability of each tuple, since they are not all the same.

Question: Assume we can still distinguish cards by suit. How many 5-card combinations have 3 10s, 1 J, and 1 Q?

$$\binom{4}{3}\binom{4}{1}\binom{4}{1} = 64$$

We are going to use the unique tuple approach to calculate the probability that a 12-card hand is nice. **Fill out the following function that determines if a tuple represents a hand that is nice**. Your function must be general and work with any hand size (not just 12). You can use `test_is_nice` to verify if your implementation is correct.

In [13]:
```
def is_nice(hand_tuple):
    assert len(hand_tuple) == 13
    ### BEGIN YOUR SOLUTION
    num_odd = 0
    for count in hand_tuple:
        if count % 2 == 1:
            num_odd += 1
    return num_odd <= 1
    ### END YOUR SOLUTION
```

In [14]:
```
def test_is_nice():
    assert is_nice((1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
    assert is_nice((2, 0, 4, 0, 2, 0, 0, 1, 0, 4, 4, 0, 2))
    assert is_nice((0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0))
    assert not is_nice((2, 0, 3, 0, 2, 0, 0, 1, 0, 4, 4, 0, 2))
    assert not is_nice((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0))
    print('All tests passed')
```

In [15]:
```
test_is_nice()
```

```
All tests passed
```

Now we need to calculate the probability of a hand_tuple. However, for numerical precision, we may just want to keep track of the total number of card combinations to get a tuple. In the end, we can divide this by the number of possible 12-card hands to find the probability. **Fill in the following function that calculates this**. You can use `test_combinations` to verify your implementation is correct.

In [16]:
```
def combinations(hand_tuple):
    assert len(hand_tuple) == 13
    ### BEGIN YOUR SOLUTION
    numerator = 1
    TABLE = [1, 4, 6, 4, 1]
    for count in hand_tuple:
        numerator *= TABLE[count]
    return numerator
    ### END YOUR SOLUTION
```

In [17]:
```
def test_combinations():
    assert combinations((1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)) == 4
    assert combinations((2, 0, 3, 0, 2, 0, 0, 1, 0, 4, 4, 0, 2)) == 3
456
    assert combinations((1, 3, 3, 4, 2, 2, 1, 1, 0, 0, 0, 2, 1)) == 8
84736
    print('All tests passed')
```

In [18]:
```
test_combinations()
```

```
All tests passed
```

What remains is to write a Python generator that can iterate through all unique tuples. This is a bit tricky, but can be done with the following idea. We recursively go through each index 0 to 12 in the tuple. At each index, we can decide to add 1 to that index or move onto the next index. We just need to make sure that we only take at most 4 cards from each index and have 12 cards in total.

Complete the following function. You can use `test_unique_tuples` to verify your implementation is correct. If you wish, you may also write your own generator from scratch.

In [19]:
```
def unique_tuples(hand_size):
    def helper(index, hand_array):
        num_cards = sum(hand_array)
        if num_cards == hand_size:
            yield tuple(hand_array)
            return
        if index >= 13:
            return
        ### BEGIN YOUR SOLUTION
        if hand_array[index] < 4:
            hand_array[index] += 1
            yield from helper(index, hand_array)
            hand_array[index] -= 1
        max_cards_left = (12-index) * 4
        if hand_size - num_cards <= max_cards_left:
            yield from helper(index+1, hand_array)
        ### END YOUR SOLUTION
    yield from helper(0, [0 for x in range(13)])
```

In [20]:
```
def test_unique_tuples():
    total_combinations = 0
    total_tuples = 0
    for hand_tuple in unique_tuples(6):
        total_combinations += combinations(hand_tuple)
        total_tuples += 1
    assert total_tuples == 18395
    assert total_combinations == 20358520
    print('All tests passed')
```

In [21]:
```
test_unique_tuples()
```

```
All tests passed
```

In [22]:
```
TOTAL = 206379406870
total_combinations = 0
for hand_tuple in unique_tuples(12):
    if is_nice(hand_tuple):
        total_combinations += combinations(hand_tuple)
print(total_combinations / TOTAL)
```

```
0.00042909408134786544
```

In [ ]: