hotHandEffect

October 8, 2017

1 Investigation of "Hot-hand effect"

There is a widely held belief in basketball that some players have periods of time where they are much better shooters than normal. This can be called having a "hot hand". An example would be if a player hit 3 shots in a row, then many fans would expect that player to be more likely (than their usual percentage) to hit their next shot.

Whether this is a real effect or some kind of cognitive bias has been previously studied. For instance in 1984(83?, 85? FIX THIS), [RESEARCHERS] looked at actual shooting results of the Philadelphia 76ers, free throws of the Boston Celtics, and controlled-experiment shots of college students. Their data did not support the existence of a hot-hand effect. This question has been revisited, for instance in [PAPERS FROM 2005?? or so??] which also failed to find evidence to support the existence of the hot-hand effect. Interestingly, in 2015 [RESEARCHERS] noted that the sampling method used in previous studies was flawed. The flaw is subtle, but leads to some evidence for the hot-hand effect.

My intent is to investigate recent shooting results of NBA players with the goals of:
Looking for evidence to support (or reject) the existence of the hot-hand effect.
Understand the subtleties of the sampling flaw found by [RESEARCHERS].

1.1 Getting the data

The reason I chose to study NBA data was that I found a resource that makes shooting data easy to download for the NBA: www.nbasavant.com. I have downloaded all shooting data for 2016-2017 and placed it in the files nba_savant__.csv in the data/nba_savant folder. [Note: you can ostensibly download a .csv file of the shots data for the entire year, but the files seem to be limited to 50,000 lines which is not enough. That's why I split the data by month when downloading.]

There is some concern about the complete validity of the data. My biggest concern is the data for April 2017 seems to be incomplete. There are not near enough total shots for the month and spot checking some players shows many fewer shots than expected for that player. However, working with this data will at least provide a framework for studying similar datasets.

Before reading in the data, we will load pandas, a python library used for data analysis.

```
In [1]: import pandas as pd
```

Now, we can read in the data from the downloaded .csv files and store the data as a DataFrame (a pandas data structure).

```
In [2]: import glob
        files_to_read = glob.glob('../data/nba_savant/*.csv')

        shots = pd.concat((pd.read_csv(f) for f in files_to_read))
```

We can look at the first few rows of data to get a sense of what data we have (and if the read did what we expected it to do):

```
In [3]: shots.head()
```

Out[3]:

	name	team_name	game_date	season	espn_player_id	team_id	espn_game_id	period	minutes_remaining	seconds_remaining	...	shot_type	shot_distance	opponent	x	y	dribbles	touch_time	defender_name	defender_distance	shot_clock
0	Andre Drummond	Detroit Pistons	2017-01-01	2016	6585.0	1610612765	400899381	4	5	57	...	2PT Field Goal	0	Miami Heat	0	1	0	0.0	NaN	0.0	0.0
1	Nerlens Noel	Philadelphia 76ers	2017-01-30	2016	2991280.0	1610612755	0	3	7	18	...	2PT Field Goal	0	Sacramento Kings	0	1	0	0.0	NaN	0.0	0.0
2	Jon Leuer	Detroit Pistons	2017-01-01	2016	6452.0	1610612765	400899381	1	3	29	...	2PT Field Goal	0	Miami Heat	0	1	0	0.0	NaN	0.0	0.0
3	Dwight Howard	Atlanta Hawks	2017-01-29	2016	2384.0	1610612737	400900132	4	0	45	...	2PT Field Goal	0	New York Knicks	0	1	0	0.0	NaN	0.0	0.0
4	Andre Drummond	Detroit Pistons	2017-01-01	2016	6585.0	1610612765	400899381	1	7	57	...	2PT Field Goal	0	Miami Heat	0	1	0	0.0	NaN	0.0	0.0

[5 rows x 22 columns]

Each row of the DataFrame consists of one shot (the observation) and 22 variables. Those variables are the columns. Note that the ... indicates we are not seeing all of the columns. Pandas has a setting that gives the maximum number of columns to print. It appears that value defaults to 20. We can increase this value and then look at the first few rows again (using new max of 60, but 22 would suffice).

In [4]:
```
pd.set_option('display.max_columns', 60)
shots.head()
```

Out[4]:

	name	team_name	game_date	season	espn_player_id	team_id	espn_game_id	period	minutes_remaining	seconds_remaining	shot_made_flag	action_type	shot_type	shot_distance	opponent	x	y	dribbles	touch_time	defender_name	defender_distance	shot_clock
0	Andre Drummond	Detroit Pistons	2017-01-01	2016	6585.0	1610612765	400899381	4	5	57	1	Alley Oop Dunk Shot	2PT Field Goal	0	Miami Heat	0	1	0	0.0	NaN	0.0	0.0
1	Nerlens Noel	Philadelphia 76ers	2017-01-30	2016	2991280.0	1610612755	0	3	7	18	1	Alley Oop Dunk Shot	2PT Field Goal	0	Sacramento Kings	0	1	0	0.0	NaN	0.0	0.0
2	Jon Leuer	Detroit Pistons	2017-01-01	2016	6452.0	1610612765	400899381	1	3	29	0	Jump Shot	2PT Field Goal	0	Miami Heat	0	1	0	0.0	NaN	0.0	0.0
3	Dwight Howard	Atlanta Hawks	2017-01-29	2016	2384.0	1610612737	400900132	4	0	45	0	Jump Shot	2PT Field Goal	0	New York Knicks	0	1	0	0.0	NaN	0.0	0.0
4	Andre Drummond	Detroit Pistons	2017-01-01	2016	6585.0	1610612765	400899381	1	7	57	0	Jump Shot	2PT Field Goal	0	Miami Heat	0	1	0	0.0	NaN	0.0	0.0

```
2              0  Alley Oop Dunk Shot  2PT Field Goal             0
3              1  Alley Oop Dunk Shot  2PT Field Goal             0
4              1  Alley Oop Dunk Shot  2PT Field Goal             0

           opponent  x  y  dribbles  touch_time  defender_name  \
0       Miami Heat  0  1         0         0.0            NaN
1  Sacramento Kings  0  1         0         0.0            NaN
2       Miami Heat  0  1         0         0.0            NaN
3   New York Knicks  0  1         0         0.0            NaN
4       Miami Heat  0  1         0         0.0            NaN

   defender_distance  shot_clock
0                0.0         0.0
1                0.0         0.0
2                0.0         0.0
3                0.0         0.0
4                0.0         0.0
```

1.2 Strategy

For an initial investigation, I plan on making the following assumptions:

Each player's shots are to be investigated as one sequence throughout the entire year. A different, and perhaps more useful, choice would be to split up each players shots by game. We will look at the data split by game later.

To consider the existence of the hot-hand effect we will only look at whether the previous shot was a make or miss. The literature standard seems to be looking at shooting percentages after streaks of 1, 2, or 3 consecutive makes or misses. We will handle more complicated scenarios later.

The only variables we will consider for analyzing each shot are those that determine:

Which player took the shot (name, espn_player_id)

When the shot was taken (game_date, period, minutes_remaining, seconds_remaining) [Note: these are used to determine the order the shots occurred in.]

Whether the shot was a make or miss (shot_made_flag)

We will not, at least initially, be considering other variables such as those associated to shot difficulty, opponents, or effects of other players shooting on a given night.

1.3 Rearranging the data

Let's start by removing columns we are not interested in. Actually, we are only keeping the columns we are interested in.

```
In [5]: shots = shots[:][['name', 'game_date', 'espn_player_id', 'period', 'minutes_remaining', 'seconds
        shots.head()
```

```
Out[5]:             name   game_date  espn_player_id  period  minutes_remaining  \
        0  Andre Drummond  2017-01-01          6585.0       4                  5
        1    Nerlens Noel  2017-01-30       2991280.0       3                  7
        2       Jon Leuer  2017-01-01          6452.0       1                  3
        3   Dwight Howard  2017-01-29          2384.0       4                  0
        4  Andre Drummond  2017-01-01          6585.0       1                  7

           seconds_remaining  shot_made_flag
        0                 57               1
        1                 18               1
        2                 29               0
```

```
3                 45               1
4                 57               1
```

Now we can sort the dataframe by player and date/time. That will help us to easily find the previous shot for each player.

```
In [6]: shots = shots.sort_values(by=['espn_player_id', 'game_date','period','minutes_remaining','secon
        shots.head()
```

Out[6]:

	name	game_date	espn_player_id	period	minutes_remaining	seconds_remaining	shot_made_flag
4544	Metta World Peace	2016-10-30	25.0	1	2	38	0
189	Metta World Peace	2016-10-30	25.0	1	1	6	0
15320	Metta World Peace	2016-11-08	25.0	4	6	24	0
28798	Metta World Peace	2016-11-08	25.0	4	5	39	0
33601	Metta World Peace	2016-11-23	25.0	1	7	39	1

Now we add a boolean variable to indicate if the previous shot was a make or miss.

```
In [7]: shots['previous_shot_made_flag'] = np.where((shots['shot_made_flag'].shift(1) == 1) & (shots['es
        shots_only_previous = shots[shots['espn_player_id'] == shots['espn_player_id'].shift(1)]
```

We can now calculate the shooting percentage for each player both after a make and after a miss.

```
In [8]: player_shot_df = shots_only_previous[['previous_shot_made_flag','espn_player_id','shot_made_flag
        player_shot_df.head()
```

Out[8]:

		shot_made_flag		
		mean	sum	size
espn_player_id	previous_shot_made_flag			
25.0	0	0.285714	6	21
	1	0.000000	0	6
136.0	0	0.380435	105	276
	1	0.412429	73	177
165.0	0	0.430435	198	460

Adding columns for easier access of percentage after miss and make.

```
In [9]: player_shot_df_unstack = player_shot_df.unstack()
        player_shot_df_unstack['percent_after_miss'] = player_shot_df_unstack.loc[:]['shot_made_flag'][
        player_shot_df_unstack['percent_after_make'] = player_shot_df_unstack.loc[:]['shot_made_flag'][
        player_shot_df_unstack['num_shots'] = player_shot_df_unstack.loc[:]['shot_made_flag']['size'][0]
        player_shot_df_unstack.head()
```

Out[9]:

	shot_made_flag					
	mean		sum		size	
previous_shot_made_flag	0	1	0	1	0	1
espn_player_id						
25.0	0.285714	0.000000	6.0	0.0	21.0	6.0
136.0	0.380435	0.412429	105.0	73.0	276.0	177.0

165.0	0.430435	0.386997	198.0	125.0	460.0	323.0
272.0	0.385593	0.386667	91.0	58.0	236.0	150.0
558.0	0.285714	0.500000	2.0	3.0	7.0	6.0

espn_player_id	previous_shot_made_flag	percent_after_miss	percent_after_make	num_shots
25.0		0.285714	0.000000	27.0
136.0		0.380435	0.412429	453.0
165.0		0.430435	0.386997	783.0
272.0		0.385593	0.386667	386.0
558.0		0.285714	0.500000	13.0

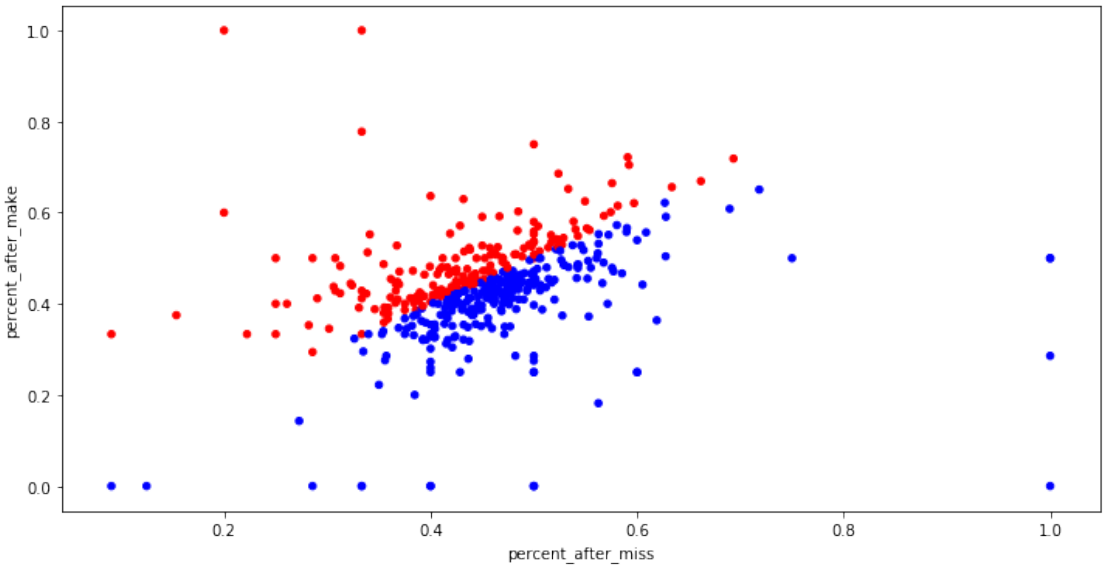
We now have the shooting percentage for every player both after a miss and after a make. If shooters do not get "hot", then it seems reasonable that for each player that percent_after_miss and percent_after_make should be roughly equal. If shooters do get "hot", then it seems reasonable that for each player that percent_after_miss should typically be less than percent_after_make.

1.4 Analyzing the data

To start investigating these questions we can first plot the data in a scatter plot. Points will be plotted with red points for a player that has a higher percentage after a make and blue points for a player that has a higher percentage after a miss.

```
In [10]: colors = np.where(player_shot_df_unstack['percent_after_miss']>player_shot_df_unstack['percent
         player_shot_df_unstack.plot(x='percent_after_miss', y='percent_after_make', kind='scatter', c=
```

```
Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa29ef95110>
```



The previous scatter plot does not seem to allow us to defnitively conclude much of anything. Players seem roughly split between those that shoot better after a miss and those that shoot better after a make.

We can run statistical tests to evaluate the results objectively.

The first test we will try is a t-test. Our null hypothesis will be that shooting percentage after a make and shooting percentage after a miss are the same. The alternative hypothesis will be that shooting percentage after a make is greater than shooting percentage after a miss. Let's assume an alpha value of 0.05 which means that we need a p-value less than 0.05 to reject the null hypothesis and conclude the data support that players have a better shooting percentage after a make. Note that this results in a one-tailed test. Annoyingly, python's standard libraries seem to only compute p-values for a two-tailed test. That means we need to divide the python calculated p-value by 2 and then possibly subtracting from 1 (depending on whether the t-statistic is positive or negative) to get our true p-value.

In [11]:
```
import scipy.stats

(t, p) = scipy.stats.ttest_1samp(player_shot_df_unstack['percent_after_make']-player_shot_df_u
if t > 0:
    # t>0 implies percent_after_make is generally greater than percent_after_miss
    # so this is the tail that (at least somewhat) supports our alternative hypothesis
    p = p/2
else:
    # t<0 implies percent_after_make is generally less than percent_after_miss
    # so this is the tail that definitely does not support our alternative hypothesis
    p = 1 - p/2
p
```

Out[11]: 0.99951561688842283

So, our p-value is (much) larger than 0.05 which means the data do not support the hot hand hypothesis. In fact, the data would have supported the hypothesis that the average shooting percentage of players is higher after a miss than after a make.

This p-value feels suspiciously high. Maybe we should check with some random data. We can fill in the shot_made_flag column with random values of 0 or 1 and see what the result is. If everything is set up correctly, then we should get an average p-value of 0.5. Note that this implies that we will want to rerun a p-value calculation multiple times, so we should create a function.

In [12]:
```
def calculatePValueForShootingPercentageAfterMake(shots):
    shots = shots.sort_values(by=['espn_player_id', 'game_date','period','minutes_remaining','
    shots['previous_shot_made_flag'] = np.where((shots['shot_made_flag'].shift(1) == 1) & (sho
    shots_only_previous = shots[shots['espn_player_id'] == shots['espn_player_id'].shift(1)]
    player_shot_df = shots_only_previous[['previous_shot_made_flag','espn_player_id','shot_mad
    player_shot_df_unstack = player_shot_df.unstack()
    player_shot_df_unstack['percent_after_miss'] = player_shot_df_unstack.loc[:]['shot_made_fl
    player_shot_df_unstack['percent_after_make'] = player_shot_df_unstack.loc[:]['shot_made_fl

    (t, p) = scipy.stats.ttest_1samp(player_shot_df_unstack['percent_after_make']-player_shot_
    if t > 0:
        # t>0 implies percent_after_make is generally greater than percent_after_miss
        # so this is the tail that (at least somewhat) supports our alternative hypothesis
        p = p/2
    else:
        # t<0 implies percent_after_make is generally less than percent_after_miss
        # so this is the tail that definitely does not support our alternative hypothesis
        p = 1 - p/2
    return p

calculatePValueForShootingPercentageAfterMake(shots)
```

Out[12]: 0.99951561688842283

Now, lets run the function on some random values and see if we get p-values spread around 0.5 or not.

```
In [13]: for x in range(20):
             shotsCopy = shots.copy()
             shotsCopy['shot_made_flag'] = np.random.randint(0,2,size=shots['shot_made_flag'].count())
             print(calculatePValueForShootingPercentageAfterMake(shotsCopy))
```

```
0.993998416438
0.999887160399
0.975065026248
0.966410361609
0.999990883012
0.999689085421
0.996097249856
0.99927941521
0.999199689557
0.983316491484
0.707874807672
0.980682535743
0.999842953492
0.849032693898
0.996173041446
0.981494862753
0.994025035981
0.990291912163
0.984775743732
0.998937696847
```

Those results suggest something is wrong in our calculations. All of the p-values from random data were greater than 0.91.

It turns out that Miller and Sanjurjo published a paper in 2015 explaining where our previous calculations went wrong. Suprisingly, from random data the expected value of the observed percentage after a make is lower than the expected value of the observed percentage after a miss. In fact, if there were n total shots, then the expected difference in the observed shooting percentages after a make and a miss is 1/(n-1).

We can adjust our function for calculating p-values accordingly.

```
In [14]: def calculatePValueForShootingPercentageAfterMake(shots):
             shots = shots.sort_values(by=['espn_player_id', 'game_date','period','minutes_remaining','s
             shots['previous_shot_made_flag'] = np.where(shots['shot_made_flag'].shift(1) == 1,1,0)
             shots_only_previous = shots[shots['espn_player_id'] == shots['espn_player_id'].shift(1)]
             player_shot_df = shots_only_previous[['previous_shot_made_flag','espn_player_id','shot_mad
             player_shot_df_unstack = player_shot_df.unstack()
             player_shot_df_unstack['percent_after_miss'] = player_shot_df_unstack.loc[:]['shot_made_fl
             player_shot_df_unstack['percent_after_make'] = player_shot_df_unstack.loc[:]['shot_made_fl
             player_shot_df_unstack['num_shots'] = player_shot_df_unstack.loc[:]['shot_made_flag']['siz

             (t, p) = scipy.stats.ttest_1samp(player_shot_df_unstack['percent_after_make']-player_shot_
             if t > 0:
                 # t>0 implies percent_after_make is generally greater than percent_after_miss
                 # so this is the tail that (at least somewhat) supports our alternative hypothesis
                 p = p/2
             else:
                 # t<0 implies percent_after_make is generally less than percent_after_miss
                 # so this is the tail that definitely does not support our alternative hypothesis
                 p = 1 - p/2
```

```
            return p

        calculatePValueForShootingPercentageAfterMake(shots)
```

```
Out[14]: 0.75880592310433004
```

The new function gives a p-value of 0.7588 which is larger than 0.05, so we still fail to conclude that NBA players' shooting percentages go up after a make.

Let's validate the new funciton against random data. Again, we expect p-values to be spread around 0.5.

```
In [15]: pValues = []
         numOver5 = 0
         for x in range(50):
             shotsCopy = shots.copy()
             shotsCopy['shot_made_flag'] = np.random.randint(0,2,size=shots['shot_made_flag'].count())
             p = calculatePValueForShootingPercentageAfterMake(shotsCopy)
             pValues.append(p)
             if p>0.5:
                 numOver5 = numOver5 + 1
             print(p)
         print "average p-value: " + str(numpy.mean(pValues))
         print "number of p-values over 0.5: " + str(numOver5)
```

```
0.875257808726
0.772559923487
0.50194759835
0.37662715207
0.565453222083
0.781052263528
0.601739882434
0.47048500954
0.45414149071
0.571537449916
0.501563061241
0.21255842209
0.221183417865
0.743536950574
0.757962110158
0.233545882647
0.953646272994
0.891953866312
0.257968487117
0.842954865031
0.543924146845
0.782133347607
0.646086756681
0.178751250802
0.0691381752039
0.515724426283
0.555238994945
0.840529770619
0.372220903577
0.667119784643
0.820249589045
0.655800947025
```

```
0.145627917143
0.246183530871
0.365313577008
0.463325590125
0.376940195083
0.790411346798
0.764942280871
0.594369723271
0.319527190788
0.71493153237
0.355313093464
0.119370800614
0.897473976511
0.529501345876
0.0756433174557
0.19307675874
0.771434737089
0.976185525554
average p-value: 0.538683313396
number of p-values over 0.5: 30
```

Our p-value calculation is validated by random data since we get an average p-value near 0.5 and roughly half of the p-values over 0.5.

Where to go from here? There are a number of things we could tidy up. Some of these include the following.

Each player's shots are viewed as one sequence for the entire year. We could split each player's shots on a per game basis.

The t-test assumes that the distribution of differences between percent after make and miss is normally distributed. We could look into the validity of that assumption.

There seem to be a fair number of outliers in the scatter plot. It seems likely that a lot of the outliers are players that took very few shots. One option to deal with the outliers would be to only consider players that took at least a certain number of shots. That seems tempting at first, but once we split each players shots into individual games we will need to consider relatively small numbers of shots anyway. Hopefully, we can use appropriate statistics to account for a player with a small number of shots. The essential problem we have is that we are using something roughly akin to an average-of-averages which is problematic with each inner average having a different sample size.

We can extend our analysis to look at more than the previous shot. Maybe the previous n shots for some n=2 or n=3.

We could account for the different shot types. Maybe do something like only looking at 3-point shots or jump shots.

We could apply machine learning (or other) techniques to look for patterns in the players that do have a better shooting percentage after a make or a miss.

1.5 Splitting shots on a per game basis

In principal we should be able to use almost the same procedure as when we split on player only. We just need to group by both the player and the game.

```
In [16]: shots_only_previous.head()
```

```
Out[16]:                    name   game_date  espn_player_id  period  \
         189    Metta World Peace  2016-10-30            25.0       1
         15320  Metta World Peace  2016-11-08            25.0       4
         28798  Metta World Peace  2016-11-08            25.0       4
         33601  Metta World Peace  2016-11-23            25.0       1
```

```
23045  Metta World Peace  2016-11-23            25.0       1

       minutes_remaining  seconds_remaining  shot_made_flag  \
189                    1                  6               0
15320                  6                 24               0
28798                  5                 39               0
33601                  7                 39               1
23045                  4                 21               0

       previous_shot_made_flag
189                          0
15320                        0
28798                        0
33601                        0
23045                        1
```

In [17]:
```
player_shot_df = shots_only_previous[['previous_shot_made_flag','game_date','espn_player_id','
player_shot_df.head()
```

Out[17]:

			shot_made_flag	
			mean	sum
espn_player_id	game_date	previous_shot_made_flag		
25.0	2016-10-30	0	0.0	0
	2016-11-08	0	0.0	0
	2016-11-23	0	1.0	2
		1	0.0	0
	2016-11-29	0	0.2	1

In [18]:
```
player_shot_df_unstack = player_shot_df.unstack()
player_shot_df_unstack['percent_after_miss'] = player_shot_df_unstack.loc[:]['shot_made_flag']
player_shot_df_unstack['percent_after_make'] = player_shot_df_unstack.loc[:]['shot_made_flag']
player_shot_df_unstack.head()
```

Out[18]:

		shot_made_flag				percent_after_miss
		mean		sum		
previous_shot_made_flag		0	1	0	1	
espn_player_id	game_date					
25.0	2016-10-30	0.0	NaN	0.0	NaN	0.0
	2016-11-08	0.0	NaN	0.0	NaN	0.0
	2016-11-23	1.0	0.0	2.0	0.0	1.0
	2016-11-29	0.2	NaN	1.0	NaN	0.2
	2016-12-02	NaN	0.0	NaN	0.0	NaN

		percent_after_make
previous_shot_made_flag		
espn_player_id	game_date	
25.0	2016-10-30	NaN
	2016-11-08	NaN
	2016-11-23	0.0
	2016-11-29	NaN
	2016-12-02	0.0

That worked, but it took a long time to evaluate. This is certainly a candidate to find a more efficient method.

There is also some concern that we are throwing away too much of our data. Consider the player with espn_player_id of 25. The splitting by game results in 3 games where the player had a miss and a make before their last shot. However, we can look at all that player's shots.

```
In [19]: shots[shots['espn_player_id']==25].groupby(['game_date']).size()
```

```
Out[19]: game_date
         2016-10-30    2
         2016-11-08    2
         2016-11-23    4
         2016-11-29    5
         2016-12-02    1
         2016-12-05    5
         2016-12-07    1
         2016-12-22    1
         2017-01-06    1
         2017-01-14    2
         2017-02-06    3
         2017-03-13    1
         dtype: int64
```

We see the player had shots in 12 games, but we are only looking at 3 of those games. This is likely worth looking at later, but we will ignore this issue for now with the logic that players not taking many shots are not what we are really interested in anyway.

We can get a summary of the data.

```
In [20]: player_shot_df.describe()
```

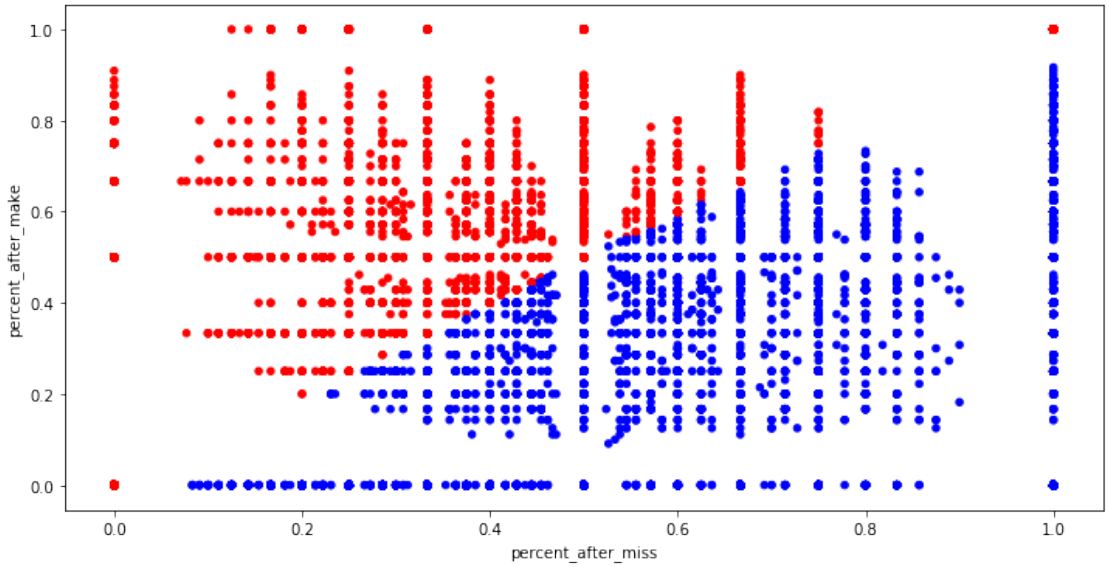
```
Out[20]:       shot_made_flag
                         mean           sum
         count   39552.000000  39552.000000
         mean        0.444909      2.070060
         std         0.297917      1.769325
         min         0.000000      0.000000
         25%         0.250000      1.000000
         50%         0.500000      2.000000
         75%         0.625000      3.000000
         max         1.000000     13.000000
```

Note that the average percentage after a miss (0.538879) is greater than the average percentage after a make (0.367874).

We can also plot the data using a scatter plot as before. Again blue dots indicate higher percentage after a miss and red dots indicate higher percentage after a make.

```
In [21]: colors = np.where(player_shot_df_unstack['percent_after_miss']>player_shot_df_unstack['percent
         player_shot_df_unstack.plot(x='percent_after_miss', y='percent_after_make', kind='scatter', c=
```

```
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa294dfb110>
```



We can also do the t-test again.

```
In [22]: (t, p) = scipy.stats.ttest_1samp(player_shot_df_unstack.percent_after_make-player_shot_df_unsta
         if t > 0:
             # t>0 implies percent_after_make is generally greater than percent_after_miss
             # so this is the tail that (at least somewhat) supports our alternative hypothesis
             p = p/2
         else:
             # t<0 implies percent_after_make is generally less than percent_after_miss
             # so this is the tail that definitely does not support our alternative hypothesis
             p = 1 - p/2
         p

Out[22]: 1.0
```

Our p-value is very close to 1. Again, we cannot conclude that players are more likely to make a shot after making the previous shot. In fact, it seems very likely players are more likely to miss a shot after making the previous shot.

```
In [23]: #shots['shot_made_flag'] = np.random.randint(0,2,size=shots['shot_made_flag'].count())
         #shots[:10]
         #df = shots[['shot_made_flag','period']].groupby(['shot_made_flag']).agg('count')
         #df.head()
```