Before you turn in the homework, make sure everything runs as expected. To do so, select **Kernel** →**Restart & Run All** in the toolbar above. Remember to submit both on **DataHub** and **Gradescope**.

Please fill in your name and include a list of your collaborators below.

In [1]:
```
NAME = "Benjamin Liu"
COLLABORATORS = ""
```

Table of Contents

In [2]:
```
from pathlib import Path
import json
import pandas as pd
import re
import numpy as np
import itertools
import matplotlib.pyplot as plt
import seaborn as sns
```

Hypothesis Testing: Does The Hot Hand Effect Exist?

Due Date: Tuesday, November 20, 2018 at 11:59pm

This homework concerns the game of basketball. If you're unfamiliar with basketball, the first minute of this youtube video (https://www.youtube.com/watch?v=wYjp2zoqQrs) does a pretty good job of giving you the basic idea.

In basketball, the "hot hands effect" is a supposed phenomenon in which a person who makes several successful baskets in a row experiences a greater probability of scoring in further attempts. For example, a player who has "made" three sucessful baskets in a row is considered to have a higher probability of making a 4th basket than if they had just missed a shot. In this assignment, we'll use 0 to represent a missed basket and 1 to represent a made basket. Restating the hot hands effect in these terms, under the hot hands theory, a player whose last three shots were `'111'` (three consecutive makes) has a higher chance of making a fourth basket than if their last three shots were `'110'` . The failed third shot "resets" their hot hands.

The notion of a hot hand is often considered to be a cognitive fallacy, a tendency for our brains to ascribe more meaning to a random sequence of shots than it rightly should. People have taken many different approaches to this topic. This homework shows how one can use statistical testing tools to test the existence of the hot hands effect in basketball.

The Data

Shot records for the Golden State Warriors (our local NBA basketball team) from the 2016-2017 season are given to you in the `data_dir` path. The files are stored in `json` format and are named '{match_date}0{team}.json'. `match_date` is the date of the game and `team` is either 'GSW' or the abbreviation for the opposing team. The structure of the data is simple: each file holds shot records for a single game in key/value pairs. The keys are player names and the values are ordered arrays of shot attempts. A `1` represents a "make" (successful attempt) and a `0` is a "miss" (failed attempt). Although this will perhaps overly simplify the analysis, for this assignment, we will not differentiate between 2-point attempts (2FGA), 3-point attempts (3FGA), and free-throws (FT).

Problem 1 [5pts]

Write a function `game_json_to_game_df` that takes a json file and builds a dataframe where each row of the table represents the information about shots for each player. Your table should have three columns `player` , `shots` , and `game` , described below:

- `player` : strings, player name
- `shots` : strings, the sequence of attempted shots concatenated into a single string e.g. '110101'.
- `game` : strings, the name of the json file (without the .json extension)

Run the cell below to see an example of the expected output. The index should just be the numbers 0 through N - 1 (i.e. you don't need to do anything special to generate the index).

In [3]:
```
pd.read_csv('single_file_shot_data_example.csv')
```

Out[3]:

	player	shots	game
0	A. Iguodala	001	201610250GSW
1	A. Varejao	01	201610250GSW
2	D. Bertans	11	201610250GSW
3	D. Dedmon	0010	201610250GSW
4	D. Green	0010011110100111	201610250GSW
5	D. Lee	110101	201610250GSW
6	D. West	10	201610250GSW
7	I. Clark	0011001000	201610250GSW
8	J. McGee	100	201610250GSW
9	J. Simmons	11111101001000001	201610250GSW
10	K. Anderson	1	201610250GSW
11	K. Durant	11110010110001001111111	201610250GSW
12	K. Leonard	011100111111100101110011001111111110	201610250GSW
13	K. Thompson	0000010110101	201610250GSW
14	L. Aldridge	01101000110111100111111000	201610250GSW
15	M. Ginobili	1001000110	201610250GSW
16	P. Gasol	1000	201610250GSW
17	P. McCaw	001	201610250GSW
18	P. Mills	001010110	201610250GSW
19	S. Curry	011111001111100000110110	201610250GSW
20	S. Livingston	010	201610250GSW
21	T. Parker	100011001	201610250GSW
22	Z. Pachulia	1	201610250GSW

Hints:

1. You can load a json file as a dictionary with:

```
with open(json_filename) as f:
     data = json.load(f)
```

2. The `json_filename` given to you is a `Path object` (https://docs.python.org/3/library/pathlib.html), which has a handy method called `stem` that you might find useful.

In [4]:
```
def game_json_to_game_df(json_filename):
    ### BEGIN Solution
    with open(json_filename) as f:
        data = json.load(f)
    df = pd.DataFrame({"player": list(data.keys()),
                       "shots": list(data.values())})
    df['shots'] = df['shots'].map(lambda xs: "".join(str(x) for x in xs))
    df['game'] = json_filename.stem
    return df
    ### END Solution
# raise NotImplementedError()
```

In [5]:
```
datafile_path = Path('data/2017/201610250GSW.json')
student_output_201610250GSW = game_json_to_game_df(datafile_path)
assert student_output_201610250GSW.shape == (23, 3), \
'The dimensions of your data frame are incorrect'
assert 'player' in student_output_201610250GSW.columns.values, \
'You seem to be missing the player column'
assert 'shots' in student_output_201610250GSW.columns.values, \
'You seem to be missing the shots column'
assert 'game' in student_output_201610250GSW.columns.values, \
'You seem to be missing the game column'
expected_output_201610250GSW = pd.read_csv('single_file_shot_data_example.csv')
assert(student_output_201610250GSW.equals(expected_output_201610250GSW))
```

Problem 2 [5pts]

Read in all 99 `json` files and combine them into a single data frame called `unindexed_shot_data`. This dataframe should have the exact same structure as in the previous part, where the index is just the numbers 0 through N - 1, where N is the total number of rows in ALL files. The following cell shows the first 25 rows of the result you should generate.

Hints:

1. The `ignore_index` property of the `append` method of the `DataFrame` class might be useful.
2. The `glob` method of the `Path` class might be useful.

In [6]:
```
pd.read_csv('every_file_shot_data_first_25_rows.csv')
```

Out[6]:

	player	shots	game
0	A. Iguodala	001	201610250GSW
1	A. Varejao	01	201610250GSW
2	D. Bertans	11	201610250GSW
3	D. Dedmon	0010	201610250GSW
4	D. Green	0010011110100111	201610250GSW
5	D. Lee	110101	201610250GSW
6	D. West	10	201610250GSW
7	I. Clark	0011001000	201610250GSW
8	J. McGee	100	201610250GSW
9	J. Simmons	11111101001000001	201610250GSW
10	K. Anderson	1	201610250GSW
11	K. Durant	1111001011000100111111	201610250GSW
12	K. Leonard	0111001111110010111001100111111110	201610250GSW
13	K. Thompson	0000010110101	201610250GSW
14	L. Aldridge	0110100011011110011111000	201610250GSW
15	M. Ginobili	1001000110	201610250GSW
16	P. Gasol	1000	201610250GSW
17	P. McCaw	001	201610250GSW
18	P. Mills	001010110	201610250GSW
19	S. Curry	0111110011111000000110110	201610250GSW
20	S. Livingston	010	201610250GSW
21	T. Parker	100011001	201610250GSW
22	Z. Pachulia	1	201610250GSW
23	A. Davis	111011000011001110110110111100100111100111001	201610280NOP
24	A. Iguodala	0101110	201610280NOP

In [7]:
```
data_dir = Path('data/2017')
### BEGIN Solution
df = pd.DataFrame()
p = data_dir.glob('**/*')
files = [x for x in p if x.is_file()]

for file in files:
    df_temp = game_json_to_game_df(file)
    df = df.append(df_temp)
unindexed_shot_data = df
### END Solution

# YOUR CODE HERE
# raise NotImplementedError()
```

In [8]:
```
assert unindexed_shot_data.shape == (2144, 3), \
'The dimensions of shot_data are off'
assert 'shots' in unindexed_shot_data.columns.values, \
'You seem to be missing the shots column'
assert '201610250GSW' in unindexed_shot_data['game'].values, \
'201610280NOP is missing from the game column of the data frame'
assert 'K. Thompson' in unindexed_shot_data['player'].values, \
'K. Thompson is missing from the player column of the data frame'
assert len(unindexed_shot_data['shots'].values.sum()) == 22051, \
'The total number of attempts seems off'
```

Run the line of code below. It converts your integer-indexed data frame into a multi-indexed one, where the first index is `game` , and the second index is `player` .

In [9]:
```
shot_data = unindexed_shot_data.set_index(['game', 'player'])
shot_data.head(5)
```

Out[9]:

game	player	shots
201701060GSW	**A. Harrison**	1
	A. Iguodala	100
	C. Parsons	0011110
	D. Green	00111100111
	D. West	001

In [10]:
```
assert shot_data.shape == (2144, 1), \
'The dimensions of shot_data are off'
assert 'shots' in shot_data.columns.values, \
'You seem to be missing the shots column'
assert '201610250GSW' in shot_data.index.get_level_values(0), \
'201610250GSW is missing from the index'
assert 'K. Thompson' in shot_data.index.get_level_values(1), \
'K. Thompson is missing from the index'
assert len(shot_data['shots'].values.sum()) == 22051, \
'The total number of attempts seems off'
```

The Hypothesis

Our **null hypothesis** is that there is no hot hands effect, meaning that the probability of making shots do not change when a player makes several baskets in a row. In this null world, every permutation of a given shot sequence is equally likely. For example `'00111'` is just as likely as `'10101'` , `'10011'` , and `'01101'` . In a universe where hot hands exists, the first sequence would be more likely than the other three.

Often in modeling the world, we begin by specifying a simplified model just to see if the question makes sense. We've hidden some other strong assumptions (perhaps erroneously) about the shots in our model. Here are some things we are not controlling for:

- Opposing defenders affect the difficulty of a shot
- Distance affects the difficulty of a shot
- Shot types vary in difficulty (3-pointers, 2-points, free-throws)
- Team mate behavior may create more favorable scoring conditions

Understanding the Data

Recall that as good data scientists, we should strive to understand our data before we analyze it (data provenance). Let's take a look at [Klay Thompson's shooting performance from Dec. 5, 2016 versus the Indiana Pacers (https://www.basketball-reference.com/play-index/shooting.fcgi?player_id=thompkl01&year_id=2017&opp_id=IND&game_location=H)](https://www.basketball-reference.com/play-index/shooting.fcgi?player_id=thompkl01&year_id=2017&opp_id=IND&game_location=H). Klay scored 60 points in 29 minutes of playing time. For those of you unfamiliar with basketball, this is a crazy number of points to score while only being in a game for 30 minutes. In the [entire history of professional basketball (https://www.basketball-reference.com/play-index/pgl_finder.cgi?request=1&match=game&is_playoffs=N&age_min=0&age_max=99&pos_is_g=Y&pos_is_gf=Y&pos](https://www.basketball-reference.com/play-index/pgl_finder.cgi?request=1&match=game&is_playoffs=N&age_min=0&age_max=99&pos_is_g=Y&pos_is_gf=Y&pos) nobody has come close (note these records are spotty before 1983).

During this game, Klay took a total of 44 shots, landing 10/11 1 point free-throws, 13/19 2 point shots, and 8/14 3 point shots. [At least one news story (https://www.usatoday.com/story/sports/nba/warriors/2016/12/06/klay-thompson-60-points-outburst-by-the-numbers-warriors-pacers/95030316/)](https://www.usatoday.com/story/sports/nba/warriors/2016/12/06/klay-thompson-60-points-outburst-by-the-numbers-warriors-pacers/95030316/) specifically called him out as having a 'hot hand' during this game.

We'll start by looking at this game to make sure we understanding the structure of the data.

Problem 3 [1pt]

We first summarize Klay's sequence of shot results. Calculate his number of `attempts` , number of `makes` (number of successes, denoted as 1), and `accuracy` for this one game. The cell below stores Klay's shots in the game described above into the klay_example variable. Your answer should go in the cell below that.

In [11]:
```
klay_example = shot_data.loc[('201612050GSW', 'K. Thompson'), 'shots']
klay_example
```

Out[11]:
```
'11011110010111111001110111101110111101010101'
```

In [12]:
```
attempts_ex = len(klay_example)
makes_ex = len(klay_example.replace("0", ""))
accuracy_ex = makes_ex / float(attempts_ex)

# YOUR CODE HERE
# raise NotImplementedError()

print(f"""
attempts:  {attempts_ex}
makes:     {makes_ex}
accuracy:  {round(accuracy_ex, 2)}
""")
```

```
attempts:  44
makes:     31
accuracy:  0.7
```

In [13]:
```
assert attempts_ex == 44
assert makes_ex == 31
assert round(accuracy_ex, 2) == 0.7
```

We might be interested in the number of runs of various lengths that Thompson makes over the course of the game. A run of length k is defined as k consecutive successes in a row. We will include overlapping runs in our counts. For example, the shot record `'1111'` contains three runs of length 2: **11**11, 1**11**1, 11**11**).

Problem 4 [2pts]

How many runs of length 2 did Thompson make in the Dec. 5, 2016 game? To answer this question, we used a regular expression, but you're free to answer this however you'd like (with code, of course). In our regular expression we make use of [positive lookbehinds (https://docs.python.org/2/library/re.html)](https://docs.python.org/2/library/re.html) `(?<=...)` .

In [14]:
```
run_length_2 = len(re.findall(r"(?<=11)\d", klay_example))

# YOUR CODE HERE
# raise NotImplementedError()

print(f"""
Klay Thompson made {run_length_2} runs of length 2 in the game against the Indiar
""")
```

```
Klay Thompson made 19 runs of length 2 in the game against the Indiana Pacers.
```

In [15]:
```
assert run_length_2 == 19
```

Problem 5 [2pts]

How many runs of length 3?

In [16]:
```
run_length_3 = len(re.findall(r"(?<=111)\d", klay_example))

# YOUR CODE HERE
# raise NotImplementedError()

print(f"""
Klay Thompson made {run_length_3} runs of length 3 in the game against the Indiar
""")
```

```
Klay Thompson made 12 runs of length 3 in the game against the Indiana Pacers.
```

In [17]:
```
# Empty, soulless cells like these contain hidden tests
# Do not delete
```

Problem 6 [10pts]

Let's generalize the work we did above by writing a function `count_runs`. `count_runs` takes two arguments:

- `shot_sequences`: a pandas series of strings, each representing a sequence of shots for a player in a game
- `run_length`: integer, the run length to count

`count_runs` should return a pandas series, where the ith element is the number of occurrences of `run_length` in the ith sequence in `shot_sequences`.

Some example input/outputs for `count_runs` are given below:

- `count_runs(pd.Series(['111', '000', '011', '000']), 2)` should return `pd.Series([2, 0, 1, 0])`
- `count_runs(pd.Series(['1100110011']), 2)` should return `pd.Series([3])`

For convenience, `count_runs` should also work if `shot_sequences` is a single string representing a single game, e.g.

`count_runs((1100110011), 2)` should return `pd.Series([3])`

In [18]:
```
def count_runs(shot_sequences, run_length):
    """
    Counts consecutive occurences of an event

    shot_sequences: a pandas series of strings, each representing a sequence of s
    run_length: integer, the run length to count

    return: pd.Series of the number of times a run of length run_length occurred
    """

    # YOUR CODE HERE
#     raise NotImplementedError()
    reg_exp = "(?<={})".format("".join(str(1) for _ in range(run_length)))
    if isinstance(shot_sequences, str):
        shot_sequences = pd.Series(shot_sequences)
    return shot_sequences.map(lambda x: len(re.findall(reg_exp, x)))
```

In [19]:
```
assert count_runs(pd.Series(['111', '000', '011', '000']), 2).equals(pd.Series([2
'There should be 2, 0, 1, and 0 runs of length 2, respectively.'
assert count_runs(pd.Series(['1100110011']), 2).equals(pd.Series([3])), \
'There should be 1 run of length 3'
assert count_runs('000', 1).equals(pd.Series(0)), \
'There should be 0 runs of 1, and your code must support string inputs (hint: if
```

In [20]:
```
# *Leers*
```

Problem 7 [5pts]

Use `count_runs` to transform the data as follows: for each player, count the number of times they have made a run of length k where $k = 1, 2, 3, \ldots, 10$ The column names should be `str(k)` and the index be the player names. A sample of the output is given below for three players in the data. The count should be across all games played by the player across the entire dataset.

In [21]:
```
pd.read_csv('count_runs_example.csv', index_col='player')
```

Out[21]:

player	1	2	3	4	5	6	7	8	9	10
K. Thompson	950	491	251	126	62	31	13	4	1	0
S. Curry	1269	714	392	200	94	41	14	5	2	1
K. Durant	1128	695	410	243	136	80	44	24	14	7

In [22]:
```
# print(pd.read_csv('count_runs_example.csv', index_col='player').loc['K. Thomps
### BEGIN Solution
run_counts = pd.DataFrame()
data_temp = shot_data.groupby("player").agg(lambda x: ",".join(x))
for k in range(1, 10+1):
    run_counts[str(k)] = count_runs(data_temp["shots"], k)

run_counts.sort_values("1", ascending = False).head()
### END Solution

# YOUR CODE HERE
# raise NotImplementedError()
```

Out[22]:

player	1	2	3	4	5	6	7	8	9	10
S. Curry	1269	714	392	200	94	41	14	5	2	1
K. Durant	1128	695	410	243	136	80	44	24	14	7
K. Thompson	950	491	251	126	62	31	13	4	1	0
D. Green	562	256	107	45	14	7	3	1	0	0
A. Iguodala	352	176	85	36	17	9	5	2	1	0

In [23]:
```
assert pd.api.types.is_string_dtype(run_counts.index), \
'Index should consist of strings.'
assert pd.api.types.is_string_dtype(run_counts.columns), \
'Column names should be strings.'
assert run_counts.loc['A. Abrines', '1'] == 8, \
'A. Abrines should have 8 single makes.'
assert run_counts.loc['K. Thompson'].sum() == 1929, \
"The sum of K Thompson's values seems off."
```

So far, we've just been exploring the data. The `run_counts` table you built above does not provide us any sort of information about the validity of the hot hands hypothesis.

`run_counts` does seem to indicate that very long streaks are pretty rare. We'll use this as a starting point for our analysis in the next section.

Defining a Test Statistic

People who refer to "hot hands" often treat it as Justice Potter Stewart treats obscenity: "I know it when I see it." (https://en.wikipedia.org/wiki/I_know_it_when_I_see_it) As data scientists, this isn't good enough for us. Instead, we should think about how to quantify the question in an empirically verifiable way.

Unfortunately, it's not immediately clear how we might test the null hypothesis. In other hypothesis test settings like website A/B testing and drug efficacy, we have obvious choices for important and measurable outcomes to demonstrate increases in revenue or positive health impacts, respectively.

However, the hot hands is not as well-defined, so we're going to try a few things that seem to have the flavor of measuring "streakiness".

Problem 8 [10pts]

Our first attempt at a test statistic will be the length of the longest streak. We saw in the previous section that long runs were rare, so perhaps we can use the occurrence of long runs as evidence either for or against the hot hands hypothesis.

Write a function `find_longest_run` that computes this test statistics. Specifically, `find_longest_run` should takes a `pd.Series` of shot sequences and returns a `pd.Series` of the lengths of the longest make sequences (consecutive 1s) in each sequence. As with `run_counts`, for convenience, make the function work for a python string input as well.

For example:

- `find_longest_run(pd.Series(['111', '000', '011', '000']))` should return `pd.Series([3, 0, 2, 0])`
- `find_longest_run(pd.Series(['1100110011']))` should return `pd.Series([2])`
- `find_longest_run('1100110011')` should return `pd.Series([2])`

In [24]:
```python
def find_longest_run(shot_sequences):
    """
    Finds longest run in a pd.Series of shot_sequences

    shot_sequences: pd.Series (string) shot data for a set of games or a single p
       to be coerced into a pd.Series

    return: as pd.Series of the lengths of longest sequences of 1s in each game
    """
    ### BEGIN Solution
    if isinstance(shot_sequences, str):
        shot_sequences = pd.Series(shot_sequences)
    return shot_sequences.map(lambda x: max(map(len, x.replace(",", "0").split("(
    ### END Solution
    # YOUR CODE HERE
#     raise NotImplementedError()
```

In [25]:
```
assert isinstance(find_longest_run(klay_example), pd.Series), \
'The output should be a pd.Series'
assert find_longest_run(pd.Series(['111', '000', '011', '000'])).equals(pd.Serie
'The longest runs should be of length 3, 0, 2, and 0, respectively.'
assert find_longest_run(pd.Series(['1100110011'])).equals(pd.Series([2])), \
'The longest run should be of length 2.'
```

In [26]:
```
# Nothing to see here.  Move along
```

Problem 9 [10pts]

If you look at the test inputs above, you'll see that the extreme game featuring Klay Thompson scoring 60 points in 29 minutes has a longest run length of 6.

Let's try to understand whether this value for our test statistic is indicative of Klay having a hot hand during this game. To do this, we need to know how 6 stacks up as a streak compared to a player similar to Klay but who definitely does not have a hot hand effect.

How do we find data on such a player? Well, **under the null hypothesis, Klay *himself* is such a player**, and the shot record we observe is really a sequence of independent shots. This suggests a bootstrap procedure to estimate the sampling distribution of longest runs. Write a function called `bootstrap_longest_run` that simulates the sampling distribution of the `longest_run` test statistic under the null hypothesis given the shot record of a single game. For example, `bootstrap_longest_run(klay_example, 100)` should return a pandas series of longest runs for 100 simulated games, where the simulated games are bootstrapped from the Klay example.

In [27]:
```
from numpy.random import choice
def bootstrap_longest_run(game, num_iter=1):
    """
    game: string, shot sequence data for a single game
    num_iter: number of statistics to generate

    returns: num_iter statistics drawn from the bootstrapped sampling distributi
    """
    ### BEGIN Solution
    samples = []
    for _ in range(num_iter):
        sample = "".join(choice(list(game), len(game), replace=True))
        samples.append(sample)
    return find_longest_run(pd.Series(samples))
    ### END Solution
    # YOUR CODE HERE
#     raise NotImplementedError()
```

In [28]:
```
longest_run_simulations = bootstrap_longest_run(klay_example, 100)
assert isinstance(longest_run_simulations, pd.Series)
assert len(longest_run_simulations) == 100
assert longest_run_simulations.max() < 30
assert longest_run_simulations.max() >= 0
```

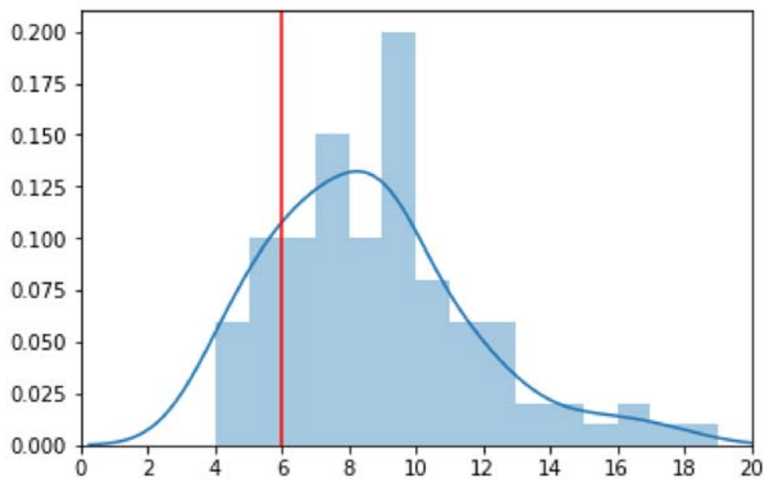
Use `bootstrap_longest_run` and the longest run statistic to answer the following question: Is Klay's performance against the Indiana Pacers indicative of hot hands? Support your answer with:

1. A plot of the observed statistic against its (bootstrapped) sampling distribution. In this plot, each possible value of longest streak length should get its own bin, centered at its value. Restrict the x-axis to the interval [0, 20].
2. A p-value compared to significance level 0.05
3. A sentence describing how the p-value should be interpreted.

In [29]:
```
# YOUR CODE HERE
# raise NotImplementedError()
### BEGIN Solution
ax = sns.distplot(longest_run_simulations, np.arange(0, 20+1, 1.0))
ax.set_xlim(0, 20)
plt.axvline(x=find_longest_run(klay_example).iloc[0], color="r")
plt.xticks(np.arange(0, 20+1, 2.0));

p_val = len(longest_run_simulations[longest_run_simulations >= 6])/len(longest_ru
print("The p-value is {}".format(p_val))
### END Solution
```

```
The p-value is 0.84
```



Answer

- 1) See avove
- 2) The p-value is higher than 0.05, hence we cannot reject the null hypothesis that Klay does not have a hot hand effect in that game
- 3) At confidence level of 95%, we cannot reject to the null hypothesis that Klay doesn't have a hot hand effect.

A Different Statistic

Arguably, the longest run isn't a particularly good test-statistic for capturing what people mean when they say "hot hands".

Let's try a test-statistics that captures the essence of "hot-hands" a bit more. We're now going to explore a well-known approach proposed by Amos Tversky (https://en.wikipedia.org/wiki/Amos_Tversky) and his collaborators. The hot hand of Tversky is similar to the notion of being "on fire" in the old arcade game NBA Jam (https://www.youtube.com/watch?v=ipzstdPtxNw). In that game, if you make 3 shots in a row with a player, your player would be on fire (with flame sprites!). While on fire (until a miss), the player has an inflated probability of making shots.

The statistic to capture this affect, called $T_{k,make}$, is easy to compute:

$$\begin{aligned} T_{k,make} &= \hat{\mathbb{P}}(\text{Make next shot} \mid \text{Made last } k \text{ shots}) \\ &= \frac{\#\{\text{Streaks of } k+1 \text{ makes in a row}\}}{\#\{\text{Streaks of } k \text{ makes in a row preceeding an attempt}\}} \end{aligned}$$

If $T_{k,make}$ is especially high, then we might say that our player is experiencing a hot hand.

A similar statistic can try to capture a cold hand reversal:

$$\begin{aligned} T_{k,miss} &= \hat{\mathbb{P}}(\text{Make next shot} \mid \text{Missed last } k \text{ shots}) \\ &= \frac{\#\{\text{Streaks of } k \text{ misses followed by make}\}}{\#\{\text{Streaks of } k \text{ misses in a row preceeding an attempt}\}} \end{aligned}$$

Note: If the value of $T_{k,miss}$ is especially high, this doesn't mean the player is expected to miss a bunch of shots in a row, instead we'd say that they tend to see reversals in their streaks.

Problem 10 [10pts]

Start by writing a utility function `count_conditionally`, which takes a `pd.Series` of shot sequence strings, a **conditioning set**, and an **event**, and returns a series of the count of the the number of times that the event follows the conditioning set in each shot sequence string.

Example Behavior 1:

If we call `count_conditionally(['111111', '01111100111'], '111', '0')`, we are counting the number of times that the event `0` follows `111` in each string. In this case, the function would return `pd.Series([0, 1])`.

Example Behavior 2:

If we call `count_conditionally(['111111', '01111100111'], '111', '1')`, we are counting the number of times that the event `1` follows `111` in each string. In this case, the function would return `pd.Series([3, 2])`. Note that events can overlap, e.g. `111111` has 3 occurrences of the event `1` that follow the condition `111`: **1111**11, 1**1111**1, 11**1111**.

As with `count_runs` and `find_longest_run`, for convenience, your `count_conditionally` function should handle a string input corresponding to a single shot sequence as well.

Hint: You should be able to recycle ideas from `count_runs`.

In [30]:
```
import regex as re
def count_conditionally(shot_sequences, conditioning_set, event='1'):
    """
    shot_sequences: pd.Series (string) of shot strings for a set of games or a s
       to be coerced into a pd.Series
    conditioning_set: string or regex pattern representing the conditioning set
    event: string or regex pattern representing the event of interest

    return: pd.Series of the number of times event occured after the
       conditioning set in each game
    """

    # YOUR CODE HERE
#     raise NotImplementedError()
    ### BEGIN Solution
    if isinstance(shot_sequences, str):
        shot_sequences = pd.Series(shot_sequences)
    return shot_sequences.map(lambda x: len(re.findall(conditioning_set+event, x
    ### END Solution
print(count_conditionally(pd.Series(['111111', '01111100111']), '111', '0'))
print(count_conditionally(pd.Series(['111111', '01111100111']), '111', '1'))
```

```
0    0
1    1
dtype: int64
0    3
1    2
dtype: int64
```

In [31]:
```
assert isinstance(count_conditionally(pd.Series(klay_example), '11'), pd.Series)
'count_conditionally should return a pd.Series'
```

In [32]:
```
# Bah, test it yourself
```

In [33]:
```
# Nobody's home
```

Worked examples

Read this section carefully. It will probably take some time to digest, but it's a very valuable lesson in statistics that we'd like you to absorb.

We'll look at the $T_{k,make}$ statistic to make sure we understand what it is, as well as what we might expect under the null vs. hot hands hypothesis.

Example 1

Let's first consider a worked out example of computing $T_{3,make}$, the observed rate of success following a streak of 3 makes. We'll use `111110001110` in our example. Looking at the string carefully, we see that the condition `111` occurs 4 times. Of the 4 occurrences, 2 are followed by a make, and 2 are followed by a miss. Thus $T_{3,make}$ for `111110001110` is 0.5. Another way of putting this is that `count_conditionally('111110001110', '111', '1')` returns the value 2 out of a possible maximum value of 4, and thus $T_{3,make}$ is 0.5.

Example 2

As another example, let's consider $T_{3,make}$ for `111110001110111`. In this case, the condition `111` occurs 5 times. However, we will not count the last `111` as a condition set, because there is no opportunity to flip again. We call this last occurrence of `111` an **unrealized conditioning set**. Of the remaining 4 occurrences, 2 are followed by a make, and 2 are followed by a miss. Thus $T_{3,make}$ for `111110001110111` is also 0.5. Another way of putting this is that `count_conditionally('111110001110111', '111', '1')` returns the value 2 out of a possible maximum value of 4, and thus $T_{3,make}$ is 0.5.

Check your understanding

Compute $T_{4,make}$ for `0000011110000111111000111` assuming the probability the player makes a shot is 75%.

▶ Click here to show the answer

Now that you know how to compute $T_{k,make}$, let's reiterate that it tells us the observed probability that we will make the next shot, given that we have made the previous k shots. That is for the sequence `0000011110000111111000111`, the fact that $T_{4,make}$ is equal to 0.6 means that the **observed probability** of making a shot after 4 shots in a row is 60%.

Computing the Expectated Value of $T_{1,make}$

Consider $T_{1,make}$, i.e. the observed probability that you make a shot, given that your last shot was also a make. Before continuing, make sure you can compute that $T_{1,make}$ of `1110` is $\frac{2}{3}$.

We ultimately want to take player shot sequences and compute $T_{k,make}$, so it'd be a good idea if we know what to expect under the null hypothesis.

Thought Exercise

Suppose that a given player's probability of making a shot is 50%, and that they make exactly 4 shots. Under the null hypothesis (hot hands does not exist), give your guess for the expected value of $T_{1,make}$. Supply your answer by setting the variable `ev_tk1_make`.

In other words, if you pick `ev_tk1_make = 0.8`, you're saying that for a shot sequence of four shots for a player with 50% accuracy, under the null hypothesis (hot hands doesn't exist) you expect that you will observe the player making 80% of their shots that follow a make.

In [34]:
```
# Doesn't matter what you write.  This is just to keep you honest about
# your intuition
ev_tk1_make = 0.5

# YOUR CODE HERE
# raise NotImplementedError()
```

In [35]:
```python
assert 0 <= ev_tk1_make <= 1
```

We're guessing that you picked `ev_tk1_make = 0.5` , which is a great guess! It seems clear that if shots are made independently, the chance of making a basket is 50%. While the OVERALL probability is 50%, the CONDITIONAL probability will not be 50%. In other words the expected value of $T_{1,make}$ will not be 0.5 under the null hypothesis if we're considering a shot sequence of 4 shots with 50% probability.

How can this be? We will show it to be true by enumerating all the possibilities. Run the cell below to list the four different possibilities for our shot sequences, with the value of $T_{1,make}$ for each sequence in the rightmost column. `n11` is how many times our conditioning set is realized and followed by a 1, and `n10` is how many times our conditioning set is realized and followed by a 0.

In [36]:
```python
def iterable_to_string(iterable):
    return ''.join(map(str, iterable))

example = pd.DataFrame({
 'sequence': [iterable_to_string(s) for s in itertools.product('10', repeat=4)]
})
example['n11'] = count_conditionally(example['sequence'], '1', '1')
example['n10'] = count_conditionally(example['sequence'], '1', '0')
example['tk1'] = (example['n11'] / (example['n11'] + example['n10'])).round(2)

example
```

Out[36]:

	sequence	n11	n10	tk1
0	1111	3	0	1.00
1	1110	2	1	0.67
2	1101	1	1	0.50
3	1100	1	1	0.50
4	1011	1	1	0.50
5	1010	0	2	0.00
6	1001	0	1	0.00
7	1000	0	1	0.00
8	0111	2	0	1.00
9	0110	1	1	0.50
10	0101	0	1	0.00
11	0100	0	1	0.00
12	0011	1	0	1.00
13	0010	0	1	0.00
14	0001	0	0	NaN
15	0000	0	0	NaN

Since each sequence is equally likely (you should prove this to yourself!), each of the possible observations for $T_{1,make}$ have the same probability, and we can just take the arithmetic average of `tk1` , dropping any undefined proportions, to get the expected value.

In [37]:
```
ev_tk1_actual = example['tk1'].dropna().mean().round(2)
print(f'The expected value of the conditional proportion is {ev_tk1_actual}')
```

```
The expected value of the conditional proportion is 0.4
```

Surprised? We certainly were! You can do a similar analysis of $T_{k,miss}$ to find that it is greater than 0.5, meaning the expected proportion of streak reversals (a 1 after a sequence of consecutive 0s) is higher than 0.5, the overall probability of getting a 1!

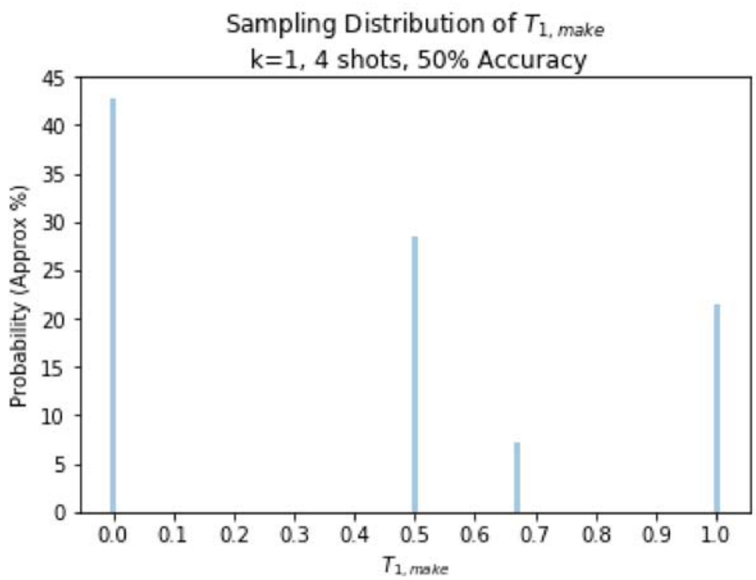
Differently put, if we label the sequence as s_1, s_2, s_3, s_4, we can write the proportion as:

$$\begin{aligned} T_{1,make} &= \hat{\mathbb{P}}(\text{Get a 1 given that previous result was 1}) \\ &= \hat{\mathbb{P}}(s_i = 1 \mid s_{i-1} = 1) \\ &= \frac{n_{11}}{n_{10} + n_{11}} \end{aligned}$$

It may seem like $\mathbb{E}\left[T_{1,make}\right]$ should really be 0.5 if the chance of making a shot is 50%, but the table above shows that this is NOT the case. The observed chance of getting a make given that you just got a make is actually 40% when you have a sequence of 4 shots.

Notice that the table above is also enough to fully describe the sampling distribution of $T_{1,make}$ for a player with an accuracy of 50%. Below is a plot of the probability distribution. There are only 4 possible values for $T_{1,make}$: $0, \frac{1}{2}, \frac{2}{3}$, and 1.

In [38]:
```
sns.distplot(example['tk1'].dropna(),
             kde=False, bins=np.arange(-0.005, 1.015, .01), norm_hist=True)
plt.xticks(np.arange(0, 1.1, 0.1))
plt.title('Sampling Distribution of $T_{1, make}$\n k=1, 4 shots, 50% Accuracy')
plt.xlabel('$T_{1,make}$')
plt.ylabel('Probability (Approx %)');
```



Problem 11 [5pts]

Recall that in the example above, we were conditioning on runs of length 1 and a player that shoots 4 times with an accuracy of 50%. Calculate the expected proportion of makes conditioned on runs of length 2 (i.e. $T_{2,make}$) when the player shoots 16 times with an accuracy of 50%.

In [39]:
```
### BEGIN Solution
example = pd.DataFrame({
 'sequence': [iterable_to_string(s) for s in itertools.product('10', repeat=16)]
})
example['n111'] = count_conditionally(example['sequence'], '11', '1')
example['n110'] = count_conditionally(example['sequence'], '11', '0')
example['tk2'] = (example['n111'] / (example['n111'] + example['n110'])).round(2

expected_proportion = example['tk2'].dropna().mean().round(2)
### END Solution

# YOUR CODE HERE
# raise NotImplementedError()
```

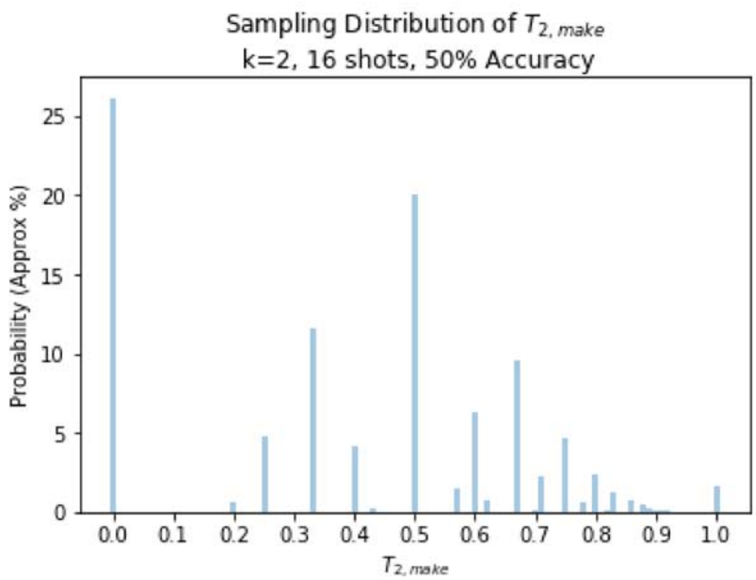
In [40]:
```
assert 0 <= expected_proportion <= 1, \
'The expected proportion should be between 0 and 1.'
```

Problem 12 [5pts]

Plot the sampling distribution of $T_{2,make}$ of a player who shoots 16 times with an accuracy of 50%. You should be able to reuse your work from the last problem.

In [41]:
```
# YOUR CODE HERE
# raise NotImplementedError()
### BEGIN Solution
sns.distplot(example['tk2'].dropna(),
             kde=False, bins=np.arange(-0.005, 1.015, .01), norm_hist=True)
plt.xticks(np.arange(0, 1.1, 0.1))
plt.title('Sampling Distribution of $T_{2, make}$\n k=2, 16 shots, 50% Accuracy'
plt.xlabel('$T_{2,make}$')
plt.ylabel('Probability (Approx %)');
expected_proportion
### END Solution
```

Out[41]: 0.40000000000000002



It turns out that the logic for calculating the exact sampling distribution for $T_{k,make}$ when a player has an accuracy other than 50% is a little more complicated than we're immediately equipped to deal with in this class. This might seem like our analysis is going to be doomed. If we can't compute $T_{k,make}$ to expect under the null hypothesis for a given player, how will be able to recognize a $T_{k,make}$ value that indicates that the hot hands hypothesis is true?

Luckily, we don't have to, because we have a tool that will allow us to approximate the sampling distribution under the null hypothesis: the bootstrap. The key observation is that the bootstrap procedure naturally preserves the player's overall shot accuracy in a given game. We'll use the bootstrap in a little while at the very end of this homework.

In short, all that hard work we just did to compute the exact sampling distribution of $T_{k,make}$ isn't going to play a role in our analysis. Nonetheless, we felt it was important to really dig in and gain intuition on this test statistic.

The "Tversky Statistic" for Hot Hand Detection

It turns out that simply measuring $T_k = T_{k,make}$ isn't as useful as the "Tversky statistic" for hot-hand detection, defined as

$$T_k = T_{k,make} - T_{k,miss}$$

The original inspiration for this statistic was to measure hot-handedness by comparing the proportion of times a player continued a success streak against their propensity to reverse a string of misses. As we saw above, computing the expected value of $T_{k,make}$ is hard and the results are counterintuitive. We're not going to formally explore the expected value of T_k, but you are free and encouraged to do so.

We will, however, mention that for reasons similar to our analysis in the previous sections, despite most people's initial intuition that the expected value of T_k should be zero, this statistic has its sampling distribution centered around a value less than 0.

Problem 13 [5pts]

The Tversky statistic is sometimes undefined (has no valid value). In our analysis, we will be discarding sequences where T_k is undefined. The reason is that it doesn't make sense to count cases where the conditioning set isn't present. Specifically describe the two cases where T_k is undefined.

Answer

- Case 1: don't have k 1s in a row in the sequence(except for the last charactor)
- Case 2: don't have k 0s in a row in the sequence(except for the last charactor)

Problem 14 [5pts]

Write a function `calc_tk_stat` that can take a `pd.Series` of shot strings and return their Tversky statistics. If the statistic is undefined, return `NaN` .

In [42]:
```
def calc_tk_stat(games, k):
    """
    Computes the tversky statistic for hot hands

    games: pd.Series (string) shot data for a set of games
    k: int, conditioning set length; number of misses/hits to condition on
    """
    ### BEGIN Solution
    tk_make = count_conditionally(games, ''.join(str(1) for _ in range(k)), '1')
            (count_conditionally(games, ''.join(str(1) for _ in range(k)), '1') 
    tk_miss = count_conditionally(games, ''.join(str(0) for _ in range(k)), '1')
            (count_conditionally(games, ''.join(str(0) for _ in range(k)), '1') 
    return pd.Series(tk_make - tk_miss)
    ### END Solution
calc_tk_stat(pd.Series(['1110100110000011']), 2)
# YOUR CODE HERE
# raise NotImplementedError()
```

Out[42]:
```
0   -0.066667
dtype: float64
```

In [43]:
```
assert np.isclose(calc_tk_stat(pd.Series(['1110100110000011']), 2), pd.Series([-
'T_2 for 1110100110000011 is -1/15'
```

Statistically Testing the Null Hypothesis

Now we return to the question of whether or not Thompson has hot hands. Under the hypothesis that he does have hot hands, Klay Thompson has a higher chance of making shots when he has recently made shots. Under the null hypothesis, his chance of making a shot is independent of recent successes.

Run the cell below, which we'll use to load all of Klay Thompson's data.

Assuming you've correctly read in `shot_data` , `klay_data` is a `pd.Series` containing Klay Thompson's shot records for the 2016-2017 season for all games (not just the game where he got 60 points).

In [44]:
```
klay_data = shot_data.loc[pd.IndexSlice[:, 'K. Thompson'], 'shots']
klay_data.head(5)
```

Out[44]:
```
game          player
201701060GSW  K. Thompson        00001101011101000001
201611010POR  K. Thompson         1011011010000010010
201701160GSW  K. Thompson      10001011000101111101111
201703020CHI  K. Thompson    0100011000100000111000000
201611100DEN  K. Thompson             0011011011100011
Name: shots, dtype: object
```

In [45]:
```
assert isinstance(klay_data, pd.Series), \
'klay_data should be a pd.Series'
assert klay_data.shape[0] == 95, \
'You have too few observations (should be 95)'
assert '0000010110101' in klay_data.values, \
'000001011010 is missing from your data'
assert klay_data.apply(lambda x: sum([int(n) for n in x])).sum() == 950, \
'You failed the checksum'
```

Problem 15 [10pts]

To help carry out the analysis at scale, write a function `calc_p_values` that can take a `pd.Series` of test statistics (one for each game) and compare it to a `pd.DataFrame` of simulated statistics. In the `pd.DataFrame`, each row corresponds to the a game, so the shape will be (number of games, number of bootstrap replications). You may assume `observed_statistics` does not contain any `NaNs`; however, `simulated_statistics` may have some.

Example Behavior

If our observed statistics are `pd.Series([0.5, 0.35, 0.4])` and our simulated statistics are a Dataframe with the values:

```
0.1, 0.3, 0.4, 0.6, 0.4, 0.6, 0.8, 0.9
0.3, NaN, 0.7, 0.1, 0.3, 0.1, 0.8, 0.6
0.3, 0.7, 0.1, 0.6, 0.7, NaN, NaN, 0.2
```

Then your function should return `pd.Series([4/8, 3/7, 3/6])`, e.g. the number of simulated statistics that matched or exceeded the observed statistic were 4 out of a possible 8.

In [46]:
```
def calc_p_values(observed_statistics, simulated_statistics):
    """
    observed_statistics: pd.Series (float), test statistics for each game
    simulated_statistics: pd.DataFrame, rows represent games, columns contain
        test statistics simulated under the null hypothesis

    return: pd.Series (float), p-values for every game between 0 and 1
    """
    # YOUR CODE HERE
#     raise NotImplementedError()
    ### BEGIN Solution
    simulated_statistics = simulated_statistics.fillna(-99)
    p_values = []
    for idx, row in simulated_statistics.iterrows():
        if np.isnan(observed_statistics.loc[idx]):
            p_values.append(np.nan)
        else :
            p_values.append(sum(row >= observed_statistics.loc[idx])/ sum(row >
    return pd.Series(p_values)
    ### END Solution
```

In [47]:
```
pv_obstat = pd.Series([0.5, 0.35, 0.4])
pv_simstat = pd.DataFrame(columns=list(range(100)), index=list(range(3)))
pv_simstat.loc[0] = pd.Series([0.1, 0.3, 0.4, 0.6, 0.4, 0.6, 0.8, 0.9])
pv_simstat.loc[1] = pd.Series([0.3, np.nan, 0.7, 0.1, 0.3, 0.1, 0.8, 0.6])
pv_simstat.loc[2] = pd.Series([0.3, 0.7, 0.1, 0.6, 0.7, np.nan, np.nan, 0.2])

assert isinstance(calc_p_values(pv_obstat, pv_simstat), pd.Series)
assert calc_p_values(pv_obstat, pv_simstat).equals(pd.Series([4/8, 3/7, 3/6]))
```

In [48]:
```
# No admittance except on party business
```

Problem 16 [Graded in the Synthesis Portion]

Carry out bootstrap hypothesis tests for all 95 records in `klay_data` for conditioning sets of length $k = 1, 2, 3$. Use 10000 bootstrap replicates to approximate the sampling distribution in each test. You will report your results in the following section. Technically, we should be worried about [multiple testing issues (https://en.wikipedia.org/wiki/Multiple_comparisons_problem)](https://en.wikipedia.org/wiki/Multiple_comparisons_problem), but you can ignore them in your analysis.

For the cell below, there is no specific structure to the output that you must produce. However, your code should compute at least:

- The observed Tversky statistic for each of the 95 games. For example, for $k = 1$, for game `'201610250GSW'`, the observed Tversky statistic is exactly `-0.250000`.
- The number of observations that had to be discarded due to an undefined Tversky statistic. For example, the game `'201610250GSW'` with shot sequence `'0000010110101'` has an undefined Tversky statistic for $k = 3$.
- The p-values for each of the 95 games. For eaxmple, for $k = 1$, for game `'201610250GSW'`, the p-value should be approximately 0.75.
- The number of games whose p-values were significant at the 5% level. For example, you might find that for $k = 1$, 90 out of 95 games have a p-value of less than 0.05, which would be strong evidence of the hot hands effect.

You'll compile the results of your findings in the next and final section of this homework.

In [49]:
```
# YOUR CODE HERE
# raise NotImplementedError()
### BEGIN Solution
def bootstrap_tk_stat(game, num_iter=10000, k=1):
    """
    game: string, shot sequence data for a single game
    num_iter: number of statistics to generate
    k:     parameter Tk

    returns: num_iter statistics drawn from the bootstrapped sampling distributi
    """
    samples = []
    for _ in range(num_iter):
        sample = "".join(choice(list(game), len(game), replace=True))
        samples.append(sample)
    return calc_tk_stat(pd.Series(samples), k)

# observed T statistic
t1_stat = calc_tk_stat(klay_data, 1)
t2_stat = calc_tk_stat(klay_data, 2)
t3_stat = calc_tk_stat(klay_data, 3)
t1_stat.index = t2_stat.index = t3_stat.index = klay_data.index

# undefined T statistics
t1_und_num = sum(t1_stat.isna())
t2_und_num = sum(t2_stat.isna())
t3_und_num = sum(t3_stat.isna())

# p-values
boot_t1 = boot_t2 = boot_t3 = pd.DataFrame()
for game in klay_data:
    boot_t1 = boot_t1.append(bootstrap_tk_stat(game, 10000, 1), ignore_index=Tru
    boot_t2 = boot_t2.append(bootstrap_tk_stat(game, 10000, 2), ignore_index=Tru
    boot_t3 = boot_t3.append(bootstrap_tk_stat(game, 10000, 3), ignore_index=Tru
boot_t1.index = boot_t2.index = boot_t3.index = klay_data.index

pv_k1 = calc_p_values(t1_stat, boot_t1)
pv_k2 = calc_p_values(t2_stat, boot_t2)
pv_k3 = calc_p_values(t3_stat, boot_t3)
pv_k1.index = pv_k2.index = pv_k3.index = klay_data.index

# check statistic significance
sig1 = sum(pv_k1 < 0.05)
sig2 = sum(pv_k2 < 0.05)
sig3 = sum(pv_k3 < 0.05)

### END Solution
```

Synthesis

Running the numerical computations in hypothesis testing is only part of the battle. Convincing others of the validity of the analysis is just as if not more important. Compile everything you have done/learned into a miniature report. Describe how you used the Tversky statistic to test whether or not Klay Thompson has hot hands. Your answer should follow the structure given below. While we can provide you with an idea of items you should definitely include in such a report, you will need to supply the wording to concisely and convincingly tell the story.

Note: DO NOT copy this cell using command mode. This will cause the autograder to fail on your notebook. You may, however, double click on the cell and copy its text.

Data Generation Model

We modeled Klay Thompson's shot record for each game as sequences of `INSERT description of random variable` with the following assumptions

- `INSERT Assumption 1`
- ...

We realize that this ignores the following real-life issues

- `INSERT Issue 1`
- ...

However, this analysis can be used as a baseline that we can compare more complicated models to.

Null Hypothesis

Our null hypothesis is `INSERT null hypothesis in plain English` . In terms of our model, this means that `INSERT mathematical implication of null hypothesis` .

Test Statistic

To test our hypothesis, we used the Tversky statistic, which can be interpreted as `INSERT plain English description in words` . This can be written mathematically as:

$$\text{INSERT LaTeX statistic} = \text{function of data}$$

Results

Looking Klay's December 5th game against the Pacers, we calculated a p-value of `INSERT p-value` for $k = 1$, which `CHOOSE ONE: is  or is not` significant at the 5% level. This can be verified visually in the following plot.

$$\text{Insert plot of sampling distribution and observed statistic}$$

We go on to analyze all of Thompson's games and find that `CHOOSE ONE: few or many` of the observations are significant at the 5% level for conditioning sets of length $k = 1, 2, 3$. The table below shows the number of observations that we discarded due to the statistic being undefined and the number that are significant at each conditioning length.

Player	Number of Games	k	Number of Games Discarded	Number of Games Significant
Thompson	95	1	`INSERT # Dropped for k=1`	`INSERT # Significant for k=1`
	-	2	`INSERT # Dropped for k=2`	`INSERT # Significant for k=2`
	-	3	`INSERT # Dropped for k=3`	`INSERT # Significant for k=3`

MY ANSWER

Data Generation Model [8pts]

We modeled Klay Thompson's shot record for each game as sequences of `binary numbers 1 means a success while 0 means a failure` with the following assumptions

- `Each games are independent experiment`
- `Each game shot have the identical distribution`

We realize that this ignores the following real-life issues

- `The shots results can depend on defense capability of opposed team`
- `The shots can depend on weather and many other factors`

However, this analysis can be used as a baseline that we can compare more complicated models to.

Null Hypothesis [5pts]

Our null hypothesis is `There is no hothand effect and chance of making a shot is independent of recent successes`. In terms of our model, this means that `Prob{success|recent successes} = Prob{success}`.

Test Statistic [2pts]

To test our hypothesis, we used the Tversky statistic, which can be interpreted as `The successful rate is the same regardless recent successes or failures`. This can be written mathematically as:

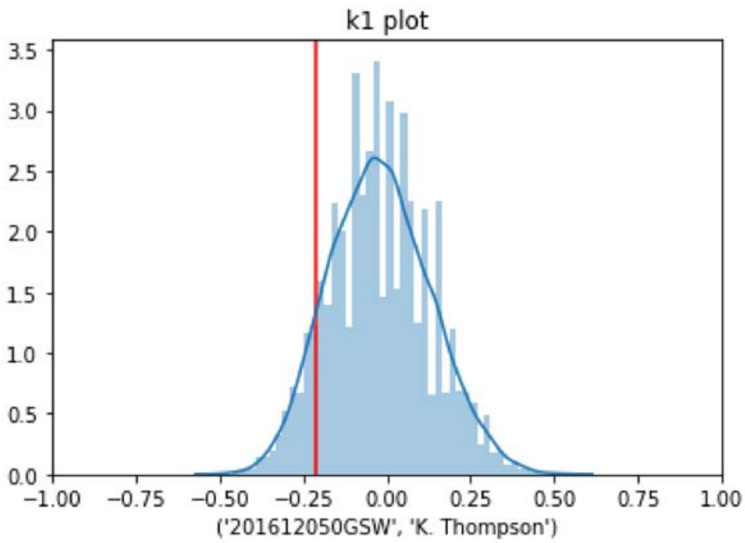
$$T_k = T_{k,make} - T_{k,miss} = 0$$

Results [20pts]

Looking Klay's December 5th game against the Pacers, we calculated a p-value of Tversky Statistics for $k = 1$, which `is` significant at the 5% level. This can be verified visually in the following plot.

Plot of sampling distribution and observed statistic

In [50]:
```
# Plotting Code
plt.axvline(x=t1_stat.loc[('201612050GSW', 'K. Thompson')], color="r")
ax = sns.distplot(boot_t1.loc[('201612050GSW', 'K. Thompson')].dropna())
plt.title("k1 plot")
ax.set_xlim(-1.0, 1.0);
# YOUR CODE HERE
# raise NotImplementedError()
```



We go on to analyze all of Thompson's games and find that `CHOOSE ONE: few or many` of the observations are significant at the 5% level for conditioning sets of length $k = 1, 2, 3$. The table below shows the number of observations that we discarded due to the statistic being undefined and the number that are significant at each conditioning length.

Player	Number of Games	k	Number of Games Discarded	Number of Games Significant
Thompson	95	1	0	5
	-	2	3	4
	-	3	42	2

In order to quickly grade your table, we ask that you include the values of the table in the cell below. `n_discarded_k*` is the number of discarded observations due to undefined statistics, and `n_sig_k*` is the number of significant observations where `*` is the length of the conditioning set.

In [51]:
```
n_discarded_k1 = t1_und_num
n_discarded_k2 = t2_und_num
n_discarded_k3 = t3_und_num
n_sig_k1 = sig1
n_sig_k2 = sig2
n_sig_k3 = sig3

print(n_discarded_k1, n_discarded_k2, n_discarded_k3)
print(n_sig_k1, n_sig_k2, n_sig_k3)

# YOUR CODE HERE
# raise NotImplementedError()
```

```
0 3 42
5 4 2
```

In [52]:
```
# No moleste
```

In [53]:
```
# Yeah I'm empty.  Wanna fight?
```

Further Reading

ESPN reports on this type of analysis
Haberstroh (2017). "He's heating up, he's on fire! Klay Thompson and the truth about the hot hand". http://www.espn.com/nba/story/_/page/presents-19573519/heating-fire-klay-thompson-truth-hot-hand-nba (http://www.espn.com/nba/story/_/page/presents-19573519/heating-fire-klay-thompson-truth-hot-hand-nba)

PDFs included in this homework folder
Daks, Desai, Goldberg (2018). "Do the GSW Have Hot Hands?"
Miller, Sanjurjo (2015). "Surprised by the Gambler's and Hot Hand Fallacies? A Truth in the Law of Small Numbers"

We thank Alon Daks, Nishant Desai, Lisa Goldberg, and Alex Papanicolaou for their contributions and suggestions in making this homework.

Submission

You're almost done!

Before submitting this assignment, ensure that you have:

1. Restarted the Kernel (in the menubar, select Kernel→Restart & Run All)
2. Validated the notebook by clicking the "Validate" button.

Then,

1. **Submit** the assignment via the Assignments tab in **Datahub**
2. **Upload and tag** the manually reviewed portions of the assignment on **Gradescope**