Who Needs Users? Just Simulate Them!

Chris Harland :: Data Scientist :: Context Relevant :: @cdubhland

(work done while at Microsoft)

Me (Data Scientist):

Not a production programmer...

Them (Devs):

"Real" programmers

Problem:

Experimentation platforms need both

I claim unit testing is a place we can all "agree"

Who needs experimentation?

Well...we do

It is how we (as in humans) establish **causality**

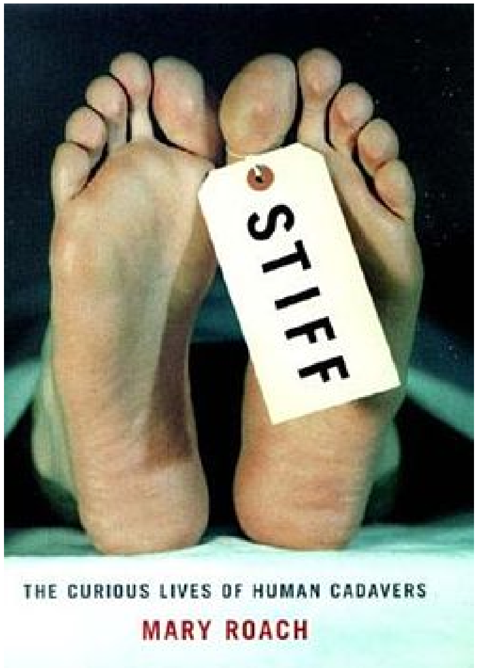
In [306]:
```
Image(filename='bloodletting.jpg', width = 300)
```

Out[306]:

The Burns Archive - Burns Archive via Newsweek, 2.4.2011

In [307]:
```
Image(filename='stiff_cover.jpg')
```

Out[307]:

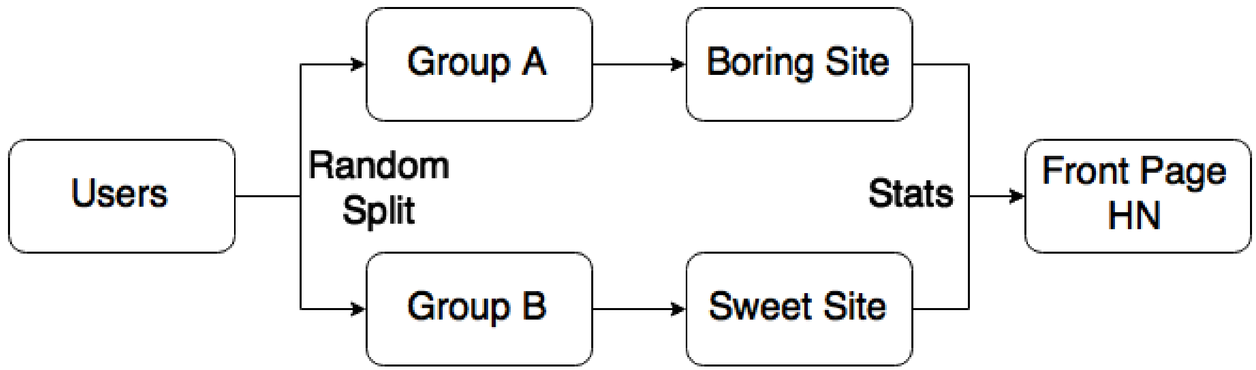


Experimentation helps us find the truth in crazy situations

A/B testing

In [406]:
```
Image(filename='A-B_testing.png')
```

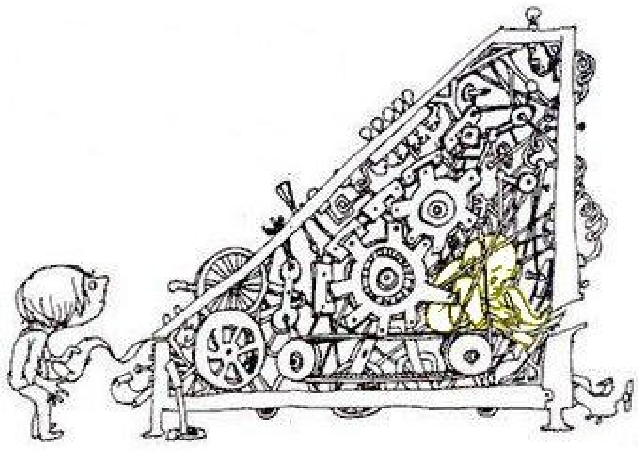
Out[406]:



A/B testing (a bit more complicated than you think...)

In [407]:
```
Image(filename='real_testing.jpg')
```

Out[407]:

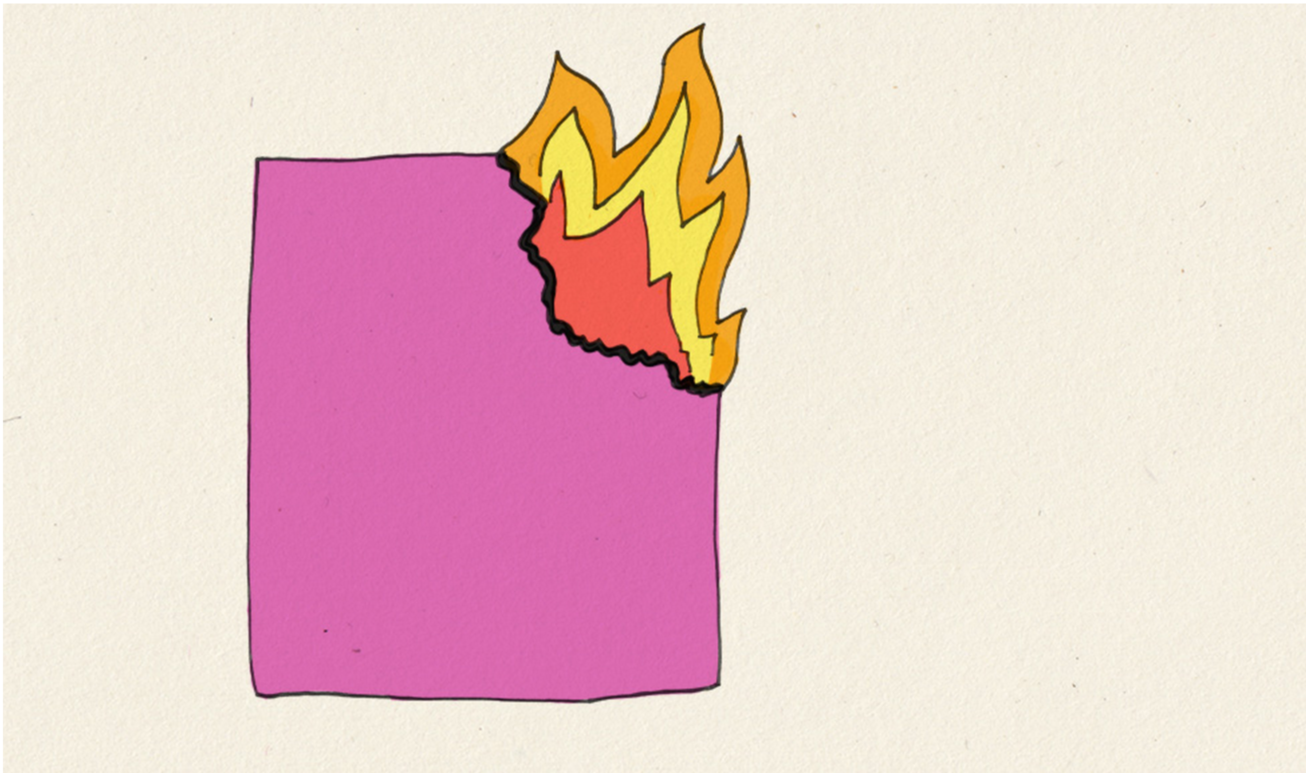


Homework Machine - A Light in the Attic - Shell Silverstein

I’m confused...

In [315]:
```
Image(filename='ab_flame_1.png', width = 700)
```

Out[315]:



How Optimizely (Almost) Got Me Fired

I'm confused...

In [316]:
```
Image(filename='ab_flame_2.png', width = 700)
```

Out[316]:

A/B testing simultaneously:

- lifts companies to the pinnacle of optimization
- is a complete waste of time and never works

A peek into my bias:

Experimentation is the story of three logs:

1. Treatment Assignment
2. Exp Platform
3. Product

Together these comprise the **execution** not the **analysis**

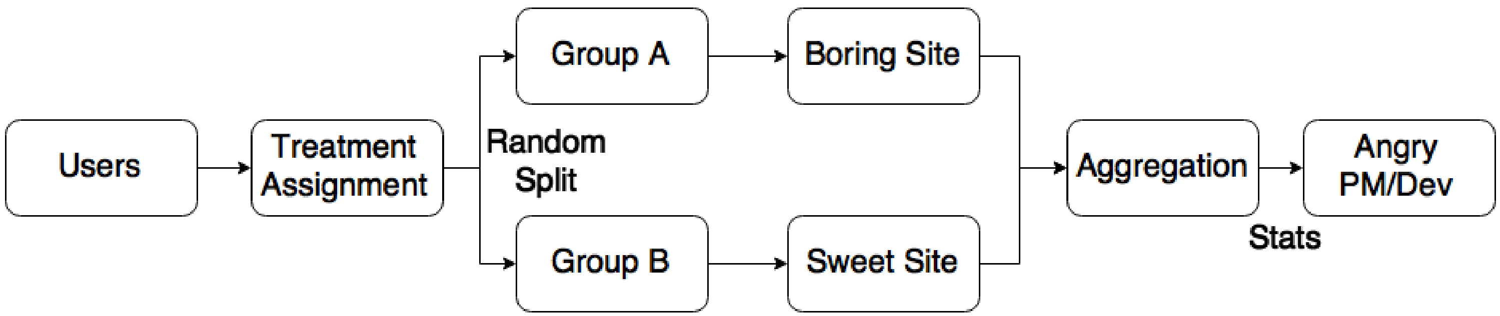
Reality : Logs :: Scientific Truth : Scorecard

So where should we start?

Assume you have a platform (of some kind) and a product (of some kind)

In [409]:
```
Image(filename='modified_ab_test.png')
```

Out[409]:



Common stumbling blocks:

- Bucketing (random numbers are hard)
- Scorecarding (counting, aggregating, and stats)

It's possible to avoid some "pitfalls"

Critical to know your platform works because users are wacky

What might a unit test look like?

In [6]:
```
# code mock up
def test_bucket_split(self):
    # Ensure that user bucketing created two equivalent groups
    for metric in self.important_metrics:
        assert abs(self.group_a[metric] - self.group_b[metric]) < self.tolerance
```

But where get these magical groups?

Make fake humans

Users are a collection of log lines

Skip the users and just get to the log lines

Anatomy of a log line:

```
1. Human visits
2. Human has choice (often influenced by treatment...you hope)
3. Human makes choice
4. (Optional) Human repeats 2 and 3 additional times
```

Logs are **generated** by a **process**

Abstract the process

```
1. Present a choice (probability distribution, computer know what these are)
2. Draw from that distribution (nice...a computer can do this)
3. Given the draw, present a second choice (another probability distribution, possibl
y different)
4. Draw again (hey a computer can do this too)
5. Repeat (oh you bet a computer can do this)
```

Simple process but it captures the essence of the log generation process

The layering of draws and choice of distributions inject flexibility and complexity

Present a choice and then make it...

In [410]:
```
# if you like python 2.7 you can high five me @cdubhland
# if you are stunned by my lack of commitment to python 3
# you can send complaints to @joelgrus
from __future__ import division
from scipy import stats
import numpy as np

def get_bernoulli_trial(p, n = 1):
    """ return a bernoulli trial of success or failure with probability p """
    return stats.bernoulli.rvs(p = p, size = n)
```

In [416]:
```
p = 0.5
n_trials = 10000
print 'Expected p ~ %0.2f and obtained p = %0.2f' % \
(p,np.mean(get_bernoulli_trial(p,n_trials)))
# We expect result to be near p
```

```
Expected p ~ 0.50 and obtained p = 0.50
```

But all decisions aren't this simple =/

Luckily math can bail us out

In [417]:
```
# We can make the probability of success a random variable
def get_beta_result(a,b, n = 1):
    """ takes a draw from beta(a,b) used to simulate random rates """
    return stats.beta.rvs(a,b, size = n)

# We can model a collection of user behaviors
def get_expon_result(mu, _lambda, n = 1):
    """ takes a draw from a exponential(mu, lambda) """
    return stats.expon.rvs(mu, _lambda, size = n)

# We can model the collective results of many choices
def get_exp_result(n,p, size = 1):
    """ return the outcome of n bernoulli trials with probability p """
    return stats.binom.rvs(n = n, p = p, size = size)

# Maybe the users visit at different frequencies
def gen_user_visit_freq(n_users = 100, _lambda = 2):
    """ return the total number of visits in a set time delta for the number of given
users """
    return stats.poisson.rvs(mu = _lambda, size = n_users)
```

Simple user click stream log

- Imagine a user comes to your site (this can be a probability)
- User executes a bernoulli trial with probability p
  - (where p is the "click through rate")
- If the user had a successful trial call another bernoulli trial with probabiliy q
  - where q is the conditional "conversion rate" $P(conv|user, click)$
- Log this as `Timestamp, user_id, impression, click, conversion`
  - `Timestamp` can be draw from distribution of average gap times or assigned sequential

In [357]:
```python
def gen_impression(p_imp = 1.0, p_click = 0.5, p_convert = 0.5):
    """ This function generates an impression, click, conversion based
    on probabilities defined by the input parameters """

    impression = get_bernoulli_trial(p_imp)[0]

    # Note: to speed this up would could draw all trials at once
    # and post process the results to make the outcomes conditional

    if impression == 1:
        did_click = get_bernoulli_trial(p_click)[0]
        # For now we assume only those that click can convert
        if did_click == 1:
            did_convert = get_bernoulli_trial(p_convert)[0]
        else:
            # Optionally this could be a bernoulli with a different p
            # (i.e. the base rate)
            did_convert = 0
        imp_arr = [impression, did_click, did_convert]
        return imp_arr
    else:
        return None
```

In [358]:
```python
[gen_impression() for _ in range(10)]
```

Out[358]:
```
[[1, 1, 0],
 [1, 1, 1],
 [1, 0, 0],
 [1, 0, 0],
 [1, 1, 0],
 [1, 1, 0],
 [1, 1, 0],
 [1, 1, 1],
 [1, 0, 0],
 [1, 1, 0]]
```

Let's get a tiny bit fancy and make this into a real log

In [359]:

```
import datetime

def gen_log_line(uid, t_current = datetime.datetime.now(), p_imp = 1.0, p_click = 0.5,
p_convert = 0.5):
    """ Get a log line for the given user and return with timestamp
    and impression info """

    imp = gen_impression(p_imp, p_click, p_convert)

    if imp is None:
        return None
    else:
        # add a random t_delta
        delta_sec = stats.norm.rvs(loc = 300, scale = 100)
        t_ = t_current + datetime.timedelta(0,delta_sec)
        timestamp = t_.strftime('%Y-%m-%d %I:%M:%S%p')
        log_line = [timestamp, uid] + imp
        return log_line, t_
```

In [360]:

```
gen_log_line('Trey Causey')[0]
```

Out[360]:

```
['2015-07-23 09:55:31PM', 'Trey Causey', 1, 0, 0]
```

Heck like a really real log

In [422]:

```
import hashlib
import pandas as pd

def create_hash_id(user, salt):
    """ returns a sha1 hash of user string combined with salt string """
    return hashlib.sha1(salt + '_' + repr(user)).hexdigest()

col_names = ['timestamp','user_id','impression','click','conversion']
user_hash = create_hash_id('Trey Causey', 'Spurs always let you down')
single_log = [gen_log_line(user_hash)[0] for x in range(10)]
pd.DataFrame(single_log, columns = col_names).sort('timestamp') \
.reset_index(drop = True).head()
```

Out[422]:

	timestamp	user_id	impression	click	conv
0	2015-07-23 09:54:01PM	a925ced33b93ee92d0f2f0763169363bf0429ce8	1	0	0
1	2015-07-23 09:54:09PM	a925ced33b93ee92d0f2f0763169363bf0429ce8	1	1	1
2	2015-07-23 09:55:26PM	a925ced33b93ee92d0f2f0763169363bf0429ce8	1	0	0
3	2015-07-23 09:56:32PM	a925ced33b93ee92d0f2f0763169363bf0429ce8	1	1	1
4	2015-07-23 09:57:24PM	a925ced33b93ee92d0f2f0763169363bf0429ce8	1	0	0

But wait…there's more!

In general the formula is:

- Encapsulate a behavior in a probability distribution
  - Poisson for distinct events
  - Exponential for time between those events
  - Binomial for total wins
  - Beta to make random probabilities
  - Normal because it's popular
- Chain those distributions together to form an impression
- Vary the parameters within each chain to generate diversity

The simplicity is deceptive

(this is bayesian stat testing backwards)

So what's the test?

Find the "unknown" parameters

Simulated logs are noisy instantiations of your supplied parameters

In other words, you put a number into the function and it spit out a ton of hand wavey examples

Your task (well, the dev's task) is to recover that parameter (within reason)

Revisiting the A/A unit test

In [425]:
```
# code mock up
def test_bucket_split(df_a, df_b, metric):
    # Ensure that user bucketing created two equivalent groups
    assert get_pval(df_a, df_b, metric) > 0.05

## Want to "recover" p > 0.05
```

In [426]:
```
p_click_control = 0.1
p_convert_control = 0.1
n_users = 1000
n_rows = 10000

# Going to hand wave this function
df_a = simulate_log_vectorized(n_users = n_users,
                               n_rows=n_rows,
                               p_click=p_click_control,
                               p_convert=p_convert_control,
                               strict = False)

df_b = simulate_log_vectorized(n_users = n_users,
                               n_rows=n_rows,
                               p_click=p_click_control,
                               p_convert=p_convert_control,
                               strict = False)
```

In [427]:
```
df_a.head(3)
```

Out[427]:

	timestamp	user_id	impression	click	conver
0	2015-07-23 22:54:32	36d1aa3c1f2e3d70773a515fb8e25a893b1c9cc4	1	1	1
1	2015-07-23 23:02:34	bbb4ffd47f8a208b45b25fdcbe19621c54d3e708	1	0	0
2	2015-07-23 23:06:50	03620c311efe60f0c5f2ecf3a6527c74f15ac3e1	1	0	0

In [428]:
```
df_b.head(3)
```

Out[428]:

	timestamp	user_id	impression	click	conve
0	2015-07-23 22:51:28	49f321d4e896801e89017eae82bfb38e7f2f4453	1	0	0
1	2015-07-23 22:58:24	2ca19d39f8a02e3fad0be9fc235e4c26e3134f58	1	0	0
2	2015-07-23 22:59:15	52ea203591b61224d62293ec387985efa722f54f	1	0	0

Sweet no `AssertionError`

In [430]:
```
test_bucket_split(df_a, df_b, 'click')
```

Why am I so wary of random number generators?

In [488]:
```
Image(filename = 'reagan.jpg', width = 500)
```

Out[488]:

But now let’s make it ”real”

In [434]:

```
# We often test many metrics
def add_metrics(n_metrics, df_a, df_b):
    for i in range(n_metrics):
        p = np.random.rand()
        # same p for both groups...should be equal
        df_a.loc[:,'metric_%d'%i] = get_bernoulli_trial(p = p, n = len(df_a))
        df_b.loc[:,'metric_%d'%i] = get_bernoulli_trial(p = p, n = len(df_b))
    return df_a, df_b

# We can make a factory of fails
def aa_fail_o_tron(df_a, df_b, n_metrics):
    # add some metrics to the pile
    df_a_mod, df_b_mod = add_metrics(n_metrics, df_a, df_b)

    # Check that all metrics come back not significant
    for i in range(n_metrics):
        test_bucket_split(df_a_mod, df_b_mod, 'metric_%d' % i)
```

In [441]:
```
aa_fail_o_tron(df_a, df_b, 10) # won't always fail
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-441-285f47eed67d> in <module>()
----> 1 aa_fail_o_tron(df_a, df_b, 10) # won't always fail

<ipython-input-434-2600a27db96f> in aa_fail_o_tron(df_a, df_b, n_metrics)
     15     # Check that all metrics come back not significant
     16     for i in range(n_metrics):
---> 17         test_bucket_split(df_a_mod, df_b_mod, 'metric_%d' % i)

<ipython-input-425-20bf50b24619> in test_bucket_split(df_a, df_b, metric)
      2 def test_bucket_split(df_a, df_b, metric):
      3     # Ensure that user bucketing created two equivalent groups
----> 4     assert get_pval(df_a, df_b, metric) > 0.05
      5 
      6 ## Want to "recover" p > 0.05

AssertionError: 
```

In [442]:

```
# But how often does it fail?
def count_dem_fails(df_a, df_b, n_metrics):
    pvals = []
    # add some metrics to the pile
    df_a_mod, df_b_mod = add_metrics(n_metrics, df_a, df_b)

    # Check that all metrics come back not significant
    for i in range(n_metrics):
        pvals.append(get_pval(df_a_mod, df_b_mod, 'metric_%d' % i))

    return sum([pval < 0.05 for pval in pvals])
```

In [445]:

```
print [count_dem_fails(df_a, df_b, 2) for _ in range(10)]
print [count_dem_fails(df_a, df_b, 5) for _ in range(10)]
print [count_dem_fails(df_a, df_b, 20) for _ in range(10)]
```

```
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 1, 1, 0]
[0, 1, 2, 1, 1, 2, 2, 1, 1, 0]
```

Be a good coder...pass those tests

In [460]:
```
def bonferroni_correction(pval, n_metrics):
    # Simply make it harder to fail by lowering pvail
    return pval / n_metrics
```

In [461]:
```
# code mock up
def test_bucket_split(df_a, df_b, metric, n_metrics = 1):
    # Ensure that user bucketing created two equivalent groups
    assert get_pval(df_a, df_b, metric) > bonferroni_correction(0.05, n_metrics)

def aa_fail_o_tron(df_a, df_b, n_metrics):
    # add some metrics to the pile
    df_a_mod, df_b_mod = add_metrics(n_metrics, df_a, df_b)

    # Check that all metrics come back not significant
    for i in range(n_metrics):
        test_bucket_split(df_a_mod, df_b_mod, 'metric_%d' % i, n_metrics)
```

In [462]:
```
aa_fail_o_tron(df_a, df_b, 10) # this passes like a lot
```

Quickly on aggregation

How should we properly aggregate raw logs before hitting them with stats stick?

Example:

How do you calculate the **average page click rate per user**?

I see alot of this:

```
df.click.mean()
```

Don't do that

Again…real log stuff looks more like

In [474]:
```
df_heavy_users = simulate_log_vectorized(n_users = 10,
                                         n_rows= 50000,
                                         p_click=0.8,
                                         p_convert=0.1,
                                         strict=False)

df_light_users = simulate_log_vectorized(n_users = 1000,
                                         n_rows= 10000,
                                         p_click=0.1,
                                         p_convert=0.1,
                                         strict=False)

df_users = pd.concat([df_heavy_users, df_light_users])
```

In [475]:
```
print 'Impression level click average: %0.3f' % df_users.click.mean()
print 'User level click average: %0.3f' % df_users.groupby('user_id').click.mean().mea
n()
```

```
Impression level click average: 0.684
User level click average: 0.108
```

Oh but it gets important

Let's say your awesome experiment lifts heavy users CTR ~10%

In [482]:
```
df_heavy_users_moved = simulate_log_vectorized(n_users = 10,
                                               n_rows= 50000,
                                               p_click=0.88,
                                               p_convert=0.1,
                                               strict=False)

df_users_moved = pd.concat([df_heavy_users_moved, df_light_users])
```

In [483]:
```
print 'Impression level click average: %0.3f' % df_users_moved.click.mean()
print 'User level click average: %0.3f' % df_users_moved.groupby('user_id').click.mea
n().mean()
```

```
Impression level click average: 0.752
User level click average: 0.109
```

In [484]:
```
click_report(df_users, df_users_moved)
```

```
Impression level control average: 0.684
Impression level treatment average: 0.752
Lift: 0.10

User level control click average: 0.108
User level treatment click average: 0.109
Lift: 0.01
```

Works the other way too

Let's say 10% of your light users exhibited a 10% lift

In [485]:

```
df_heavy_users = simulate_log_vectorized(n_users = 10,
                                         n_rows= 50000,
                                         p_click=0.8,
                                         p_convert=0.1,
                                         strict=False)

df_stubborn_light_users = simulate_log_vectorized(n_users = 900,
                                                  n_rows= 9000,
                                                  p_click=0.1,
                                                  p_convert=0.1,
                                                  strict=False)

df_cooperative_light_users = simulate_log_vectorized(n_users = 100,
                                                     n_rows= 1000,
                                                     p_click=0.11,
                                                     p_convert=0.1,
                                                     strict=False)


df_users_le_sigh = pd.concat([df_heavy_users,
                              df_stubborn_light_users,
                              df_cooperative_light_users])
```

In [486]:
```
click_report(df_users, df_users_le_sigh)
```

```
Impression level control average: 0.684
Impression level treatment average: 0.684
Lift: 0.00

User level control click average: 0.108
User level treatment click average: 0.110
Lift: 0.02
```

Impression level rollups aren't sensitive enough =/

Unit test for sensitivity (too much or too little)

Avoid making ship mistakes

In [1]:
```python
def correct_rollup(df, injected_lift, ratio):
    # ratio - fraction of users effected by injected_lift

    # calculate the user level lift
    user_lift = user_level_lift(df)
    impression_level_lift = impression_level_lift(df)

    # Example case: ratio < 1.0 and injected_lift >= 0.10
    assert user_lift < impression_level_lift
```

Some stuff to think about

- Unit testing is common ground
- Log simulation is surprisingly accurate and useful
- A/B testing pitfalls lurk in every part of your stack (fear monger)
- Users are wacky...prepare yourself for them

Who Needs Users? Just Simulate Them!

Chris Harland :: Data Scientist :: Context Relevant :: @cdubhland