```
[1]: from matplotlib.pyplot import *
%matplotlib inline

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
# make figures bigger for in-class:
#import matplotlib
#font = {'size':16}
#matplotlib.rc('font', **font)
#matplotlib.rc('figure', figsize=(7.0, 4.6667))
```

DS1 Lecture 04 notebook

Jim Bagrow

Simulating the user/product rating process

Simulating product ratings

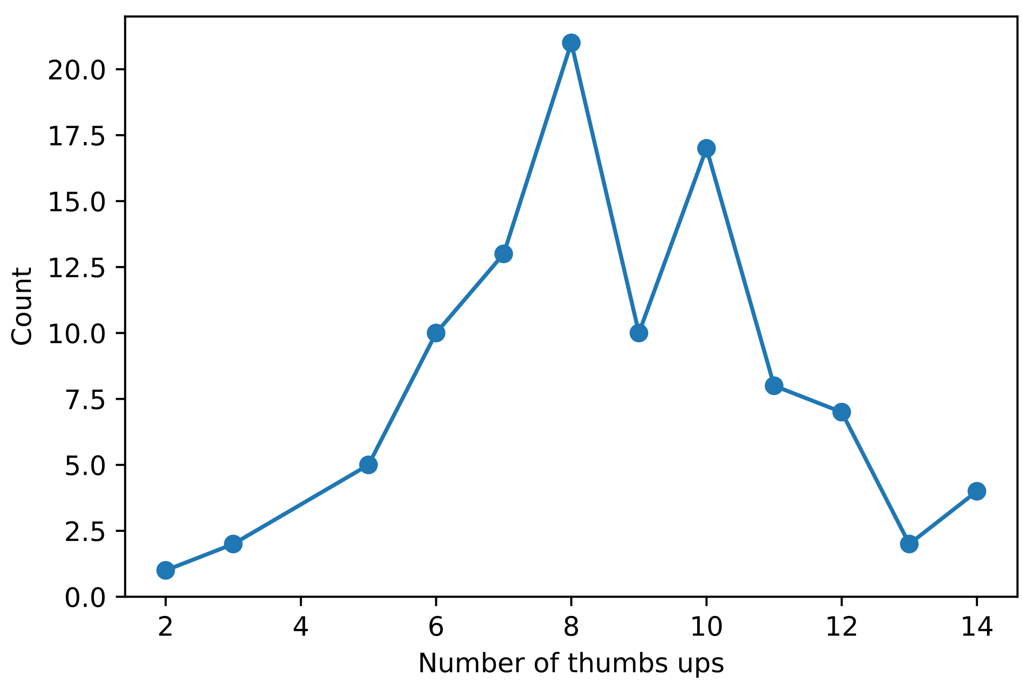
Let's simulate a group of $n = 30$ users rating a product 100 times:

```
[2]: import numpy as np

def binom(n,p):
    """return number of 'successes' out of n trials where the
    probability for a single trial to be successful is p
    """
    k = np.random.binomial(n,p)
    return k


num_simulations = 100
n_users = 30
p = 0.3 # remember: invisible to us!
results = []
for draw in range(num_simulations):
    k = binom(n_users,p)
    results.append(k)

print("Number of thumbs ups (t.u.s):")
print(results[:20], "...")
print(sum(results), p*num_simulations*n_users)
```

```
Number of thumbs ups (t.u.s):
[9, 7, 9, 14, 7, 6, 10, 10, 13, 14, 6, 12, 9, 8, 9, 6, 7, 10, 7, 7] ...
866 900.0
```

Skipping over some code details, here's a simple helper function to help us visualize **counts** of the number of thumbs ups for the different simulations:

[3]:
```python
def count_integers(L):
    """Takes a list L (of integers) and counts how many times each
    integer occurred.

    Returns two lists, the sorted values within L and the
    corresponding number of times each value occurred. This is
    useful for plotting.
    """

    # count each value:
    value2count = {} # or use collections.Counter()
    for v in L:
        try:
            value2count[v] += 1
        except KeyError: # never seen this v before, so initialize it
            value2count[v] = 1

    # break (value->count) into separate lists:
    list_values = sorted(value2count.keys())
    list_counts = []
    for v in list_values:
        list_counts.append(value2count[v])

    return list_values, list_counts


lv, lc = count_integers(results)
plt.plot(lv, lc, 'o-')
plt.xlabel("Number of thumbs ups") # always label your axes
plt.ylabel("Count")
plt.show() # would use plt.savefig() instead in a .py file
```



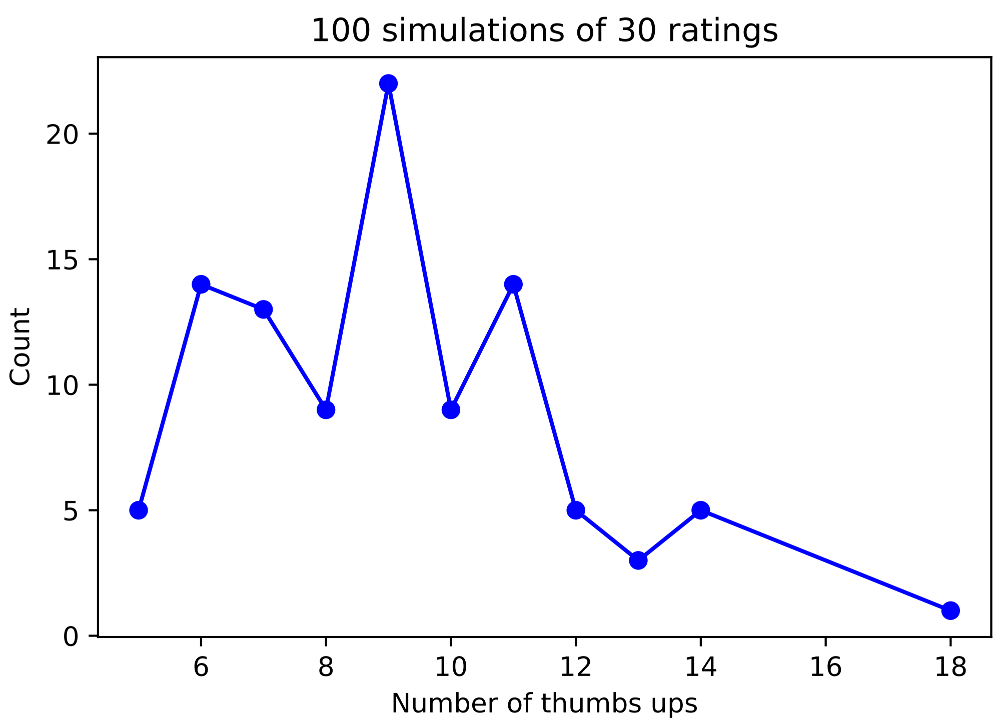
More data!

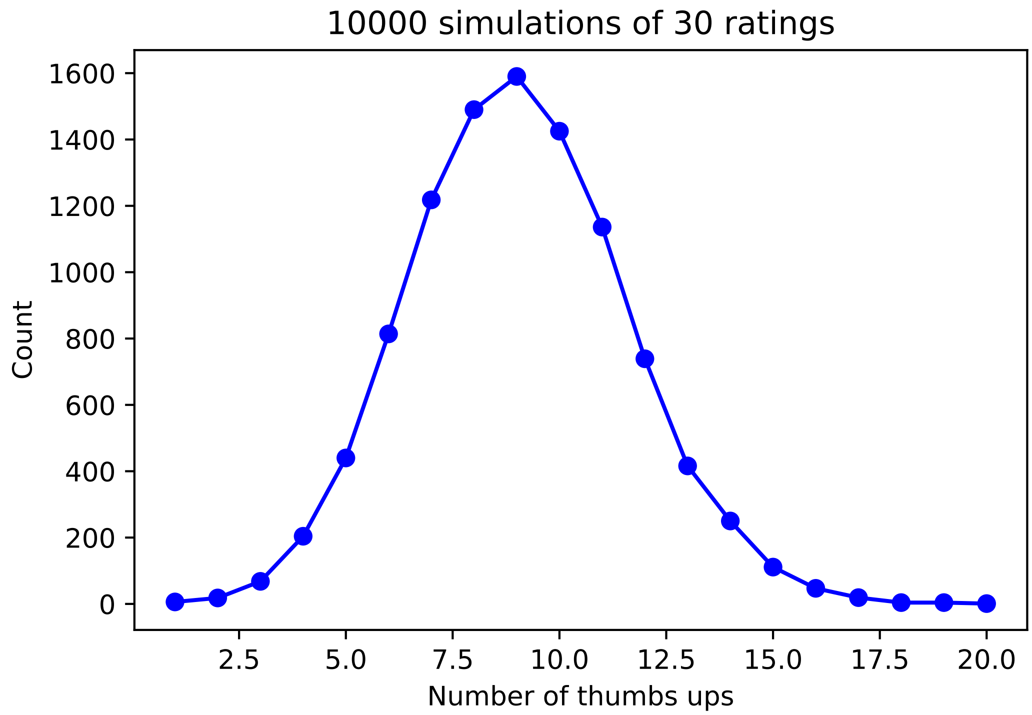
What happens if we have more simulations (for the same n and p)?

- First, some more helper functions:

[4]:
```
def simulate_experiment(num_samples, n=30, p=0.3):
    results = []
    for draw in range(num_samples):
        k = binom(n,p)
        results.append(k)
    return results


def plot_counts(L, title=None):
    lv,lc = count_integers(L)
    plt.plot(lv,lc, 'bo-')
    plt.xlabel("Number of thumbs ups")
    plt.ylabel("Count")
    if title:
        plt.title(title, fontsize='large')
    plt.show()
```

[5]:
```
n_users = 30
for num_sims in [100,10000]:
    results = simulate_experiment(num_sims, n=n_users)
    title = '{} simulations of {} ratings'.format(num_sims,n_users)
    plot_counts(results, title=title)
```





Hmm, looks much smoother with more simulations…

Move from number of thumbs up to the rating, with *fraction* of 'successes'

- Aside: In practice you would **never** define the same function multiple times in a script, but you should think of this notebook of results as a slideshow or a narrative, and not a final .py file.)

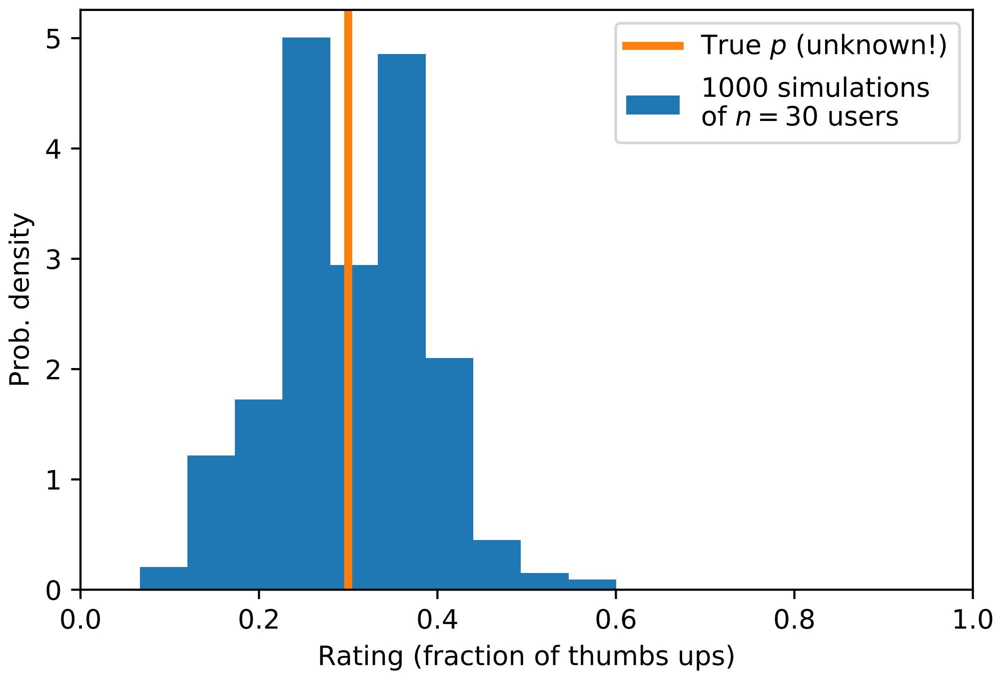
Because we are not looking at integers (# *of t.u.s*) but floats between 0 and 1 (*ratings*), let's use **histograms** [1] to visualize the distribution of ratings (be sure it's normalized: area = 1)

[1] We'll be using histograms a lot. They're a simple, powerful tool!

[6]:
```
def simulate_experiment(num_samples, n=30, p=0.3):
    results = []
    for draw in range(num_samples):
        k = binom(n,p)
        results.append(k/n) # rating not k
    return results

def plot_ratings_distribution(L, num_bins=None, label=None):
    plt.hist(L, bins=num_bins, label=label, density=True) # density!
    plt.xlim(0,1)
    plt.xlabel("Rating (fraction of thumbs ups)")
    plt.ylabel("Prob. density")
```

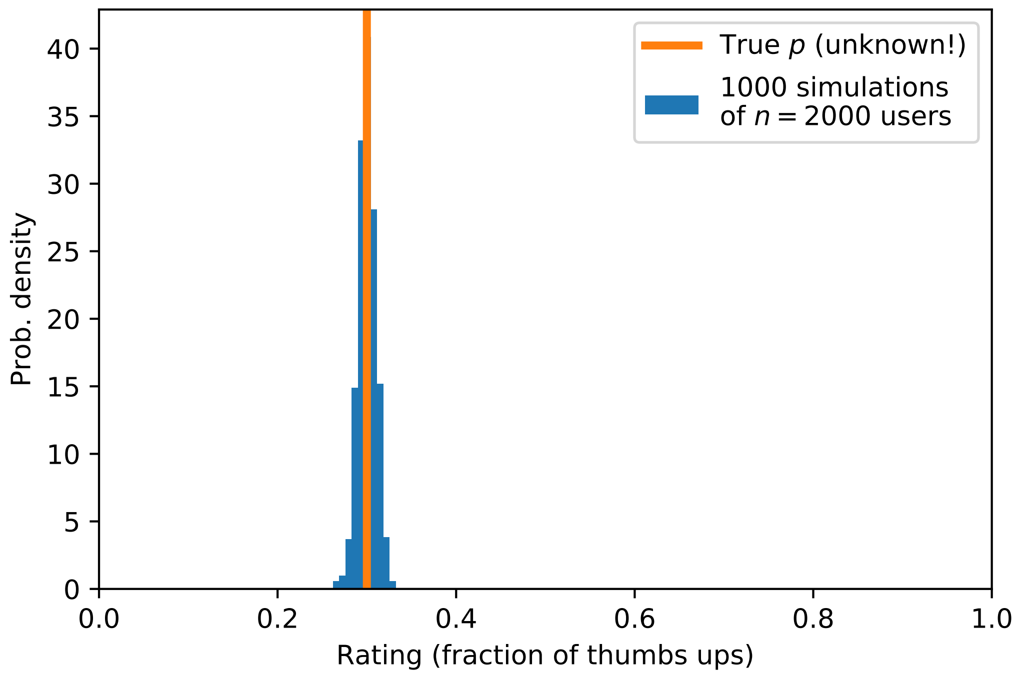
[7]:
```
num_sims = 1000
n_users = 30
```

```
p = 0.3
results = simulate_experiment(num_sims, n=n_users, p=p)

# add the secret true value as a vertical line:
plt.axvline(x=0.3, color='C1', lw=3, label='True $p$ (unknown!)')

label='{} simulations\nof $n = {}$ users'.format(num_sims, n_users)
plot_ratings_distribution(results, label=label)

plt.legend()
plt.show()
```



With more data (more users) do we get a better understanding of the true (unknown) value of p (meaning, do observed values more accurately reflect the true value)?

[8]:
```
num_sims = 1000
n_users = 2000 # up from 30
p = p
results = simulate_experiment(num_sims, n=n_users, p=p)

# add the secret true value as a vertical line:
plt.axvline(x=0.3, color='C1', lw=3, label='True $p$ (unknown!)')

label='{} simulations\nof $n = {}$ users'.format(num_sims, n_users)
plot_ratings_distribution(results, label=label)


plt.legend()
plt.show()
```



With so much more data, we should be much more certain about the value of p. Indeed, the range of observations (blue) is much more narrowly packed around the true p (orange) than the previous plot.

Compare with our models

Recall our prediction of the sampling distribution for rating.

- $E[\bar{x}] = p$
- $Var(\bar{x}) = p(1-p)/n$

Let's compare with the data:

[9]:
```
sample_mean = np.mean(results)
sample_var  = np.var(results)

print(sample_mean, p)
print(sample_var, p*(1-p)/n_users)
```

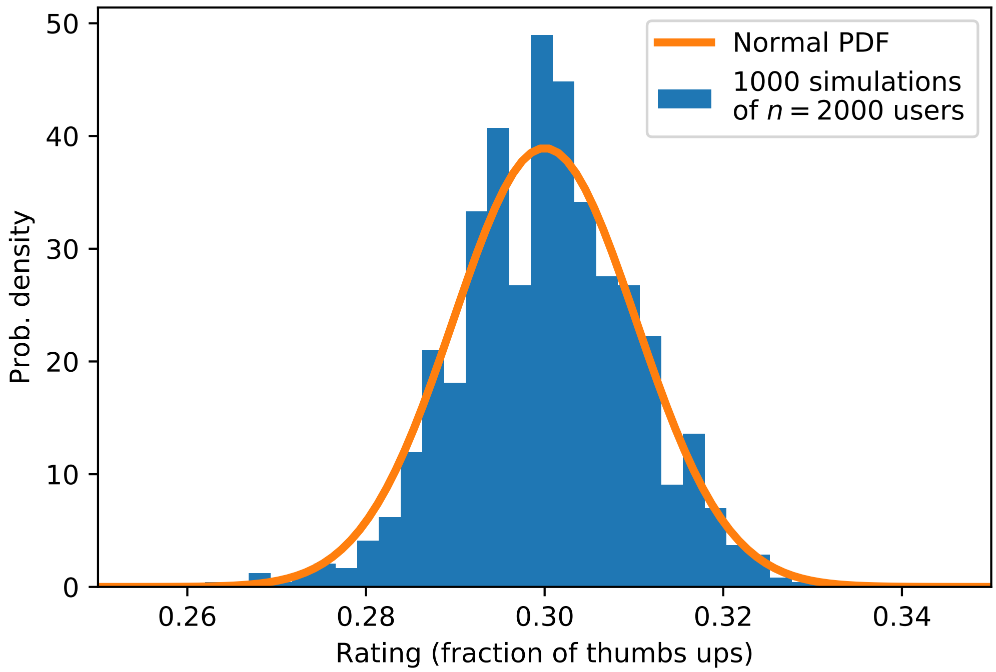
```
0.30020349999999996 0.3
9.673333775000011e-05 0.00010499999999999999
```

Pretty close!

Let's compare the data to a normal distribution with the same mean/var:

[10]:
```
import math

def normal_pdf(x, mean, var):
    bot = (2*math.pi*var)**.5
    top = np.exp(-(x-mean)**2/(2*var))
    return top/bot
```

```
# put back the data:
label='{} simulations\nof $n = {}$ users'.format(num_sims, n_users)
plot_ratings_distribution(results, num_bins='auto', label=label)
plt.xlim(0.25, 0.35)


X = np.linspace(0.25, 0.35, 100) # numpy vector!
Y = normal_pdf(X, p, p*(1-p)/n_users)
plt.plot(X,Y, lw=3, label='Normal PDF')
plt.legend()
plt.show()
```



Wow, actually **pretty close** to a normal distribution!

- This is not surprising, due to the Laplace-Demoivre theorem, a special case of the central limit theorem that shows that the binomial distribution [2] converges to a normal distribution with *lots of data*.

[2] Recall our ratings are just k/n where k is a binomial RV.

This is the Normal Approximation

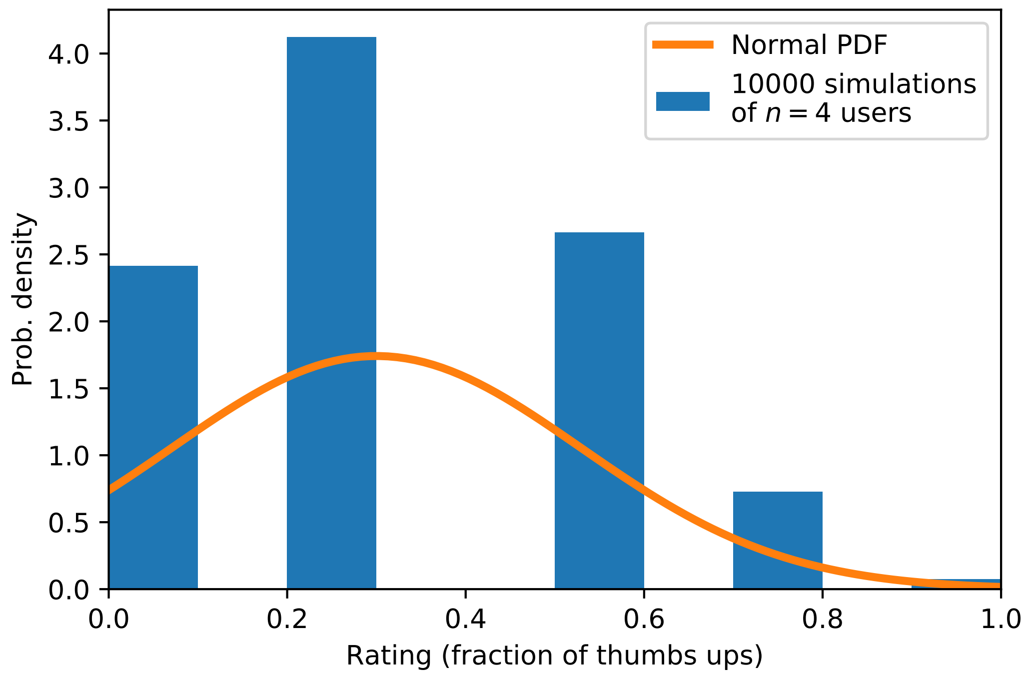
So, we can use the normal distribution with our model's mean and variance to **approximate** our distribution of ratings!!

But whenever you work with an approximation you need to think about whether it's **reasonable**.

Does this normal approximation fail?

One type of failure:

[11]:
```
num_sims = 10000
n_users = 4
p = 0.3
results = simulate_experiment(num_sims, n=n_users, p=p)

label='{} simulations\nof $n = {}$ users'.format(num_sims, n_users)
plot_ratings_distribution(results, num_bins=10, label=label)

X = np.linspace(0., 1, 100) # numpy vector!
Y = normal_pdf(X, p, p*(1-p)/n_users)
plt.plot(X,Y, lw=3, label='Normal PDF')

plt.legend()
plt.show()
```



What happened?

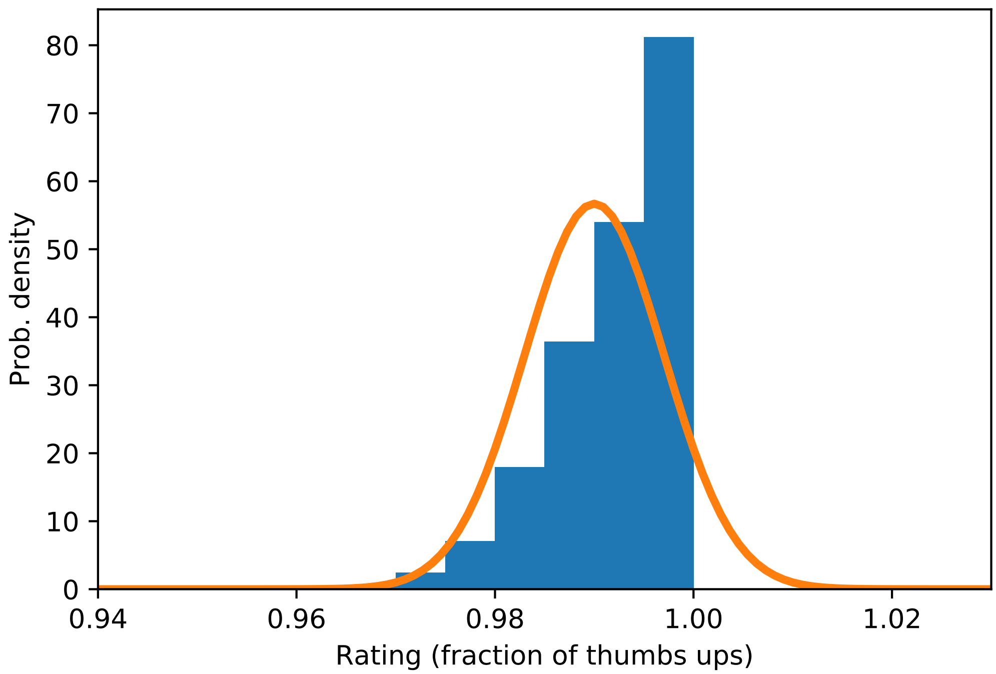
Only 4 users, so the rating "fraction" is strongly discretized:

- (0/4, 1/4, 2/4, 3/4, 4/4) are the only allowed values

Another failure:

[12]:
```
num_sims = 100000
n_users = 200
p = 0.99
results = simulate_experiment(num_sims, n=n_users, p=p)

label='{} simulations\nof $n = {}$ users'.format(num_sims, n_users)
plot_ratings_distribution(results, num_bins=10, label=label)
```

```
X = np.linspace(0.94, 1.03, 100)
Y = normal_pdf(X, p, p*(1-p)/n_users)
plt.plot(X,Y, lw=3)

plt.xlim(0.94,1.03)
plt.show()
```



What happened?

Cannot have ratings > 1. Here p is very close to 1, so we are pushed right up against the boundary. Normal distribution *doesn't account for this*.

Normall approximation best for large n and p not *too close* to 0 or 1.

What can we use the normal approximation for?

[back to board]