# (2) Hypothesis Testing

March 1, 2016

In [4]:
```
%matplotlib inline

#python includes
import sys

#standard probability includes:
import numpy as np #matrices and data structures
import scipy.stats as ss #standard statistical operations
import pandas as pd #keeps data organized, works well with data
import matplotlib
import matplotlib.pyplot as plt #plot visualization
```

In [5]:
```
##ASSIGNMENT 1 Solution. Some steps from this setup the data for our hypothesis test so they ar
## A. Read in the CSV.
file = '2015 CHR Analytic Data.csv'
if (len(sys.argv) > 1) and (sys.argv[1][-4:].lower() == 'csv'):
    file = sys.argv[1]
print "loading %s" % file
data = pd.read_csv(file,sep=',',low_memory=False)

#A.1) COLUMN HEADERS:
#print "\n (1) COLUMN HEADERS:"
valsColumn = [ currColumn for currColumn in data.columns  if "Value" in currColumn
               or "COUNTYCODE" in currColumn or "County" in currColumn]
data = data[data.COUNTYCODE != 0] #filter data frame to only county rows (those with countycode
data = data[valsColumn] #filter to "value" columns
valsColumn = [v for v in valsColumn if "Value" in v] #drop the non-value columns
#print valsColumn
#A.2) TOTAL COUNTIES IN FILE: The total number of counties in the file. (see \NOTE" above)
print "\n(2) TOTAL COUNTIES IN FILE:"
print("\t\t\t\t%d "%(len(data.index)))
#A.3) TOTAL RANKED COUNTIES: The total number of counties without a \1" in the field \County th
notRankedCounties = [ idx for idx,isRanked in enumerate(data['County that was not ranked']) if
rankedCounties = [ idx for idx,isRanked in enumerate(data['County that was not ranked']) if isRa
#print "\n(3) TOTAL RANKED COUNTIES:"
#print("\t\t\t\t%d "%(len(rankedCounties)))

## B. Model whether a county was ranked based on its population.
#B.4) HISTOGRAM OF POPULATION: a1_4_histpop.png:
#A histogram of the field \'2011 population estimate Value'". Choose an appropriate number of b

#png_file = 'a1_4_histpop.png'
#print "\n(4) HISTOGRAM OF POPULATION: %s" % png_file
```

```
import string
if data['2011 population estimate Value'].dtype != 'int':
    data['2011 population estimate Value'] = data['2011 population estimate Value'].apply(lambda
    data['2011 population estimate Value'] = data['2011 population estimate Value'].astype('int
#hist = data['2011 population estimate Value'].hist(bins=60)
#hist.get_figure().savefig(png_file)
#plt.show()

#B.5) HISTOGRAM OF LOG POPULATION: a1_5_histlog.png: Add a column, \log_pop" = log(\2011 popula
#png_file = 'a1_5_histlog.png'
#print "\n(5) HISTOGRAM OF LOG POPULATION: %s" % png_file
data['log_pop'] = np.log(data['2011 population estimate Value'])
#lhist = data['log_pop'].hist(bins=30)
#lhist.get_figure().savefig(png_file)
#plt.show()

#B.6) KERNEL DENSITY ESTIMATES: a1_6_KDE.png: 2 kernel density plots based on log_pop: (a) coun
#png_file = 'a1_6_KDE.png'
#print "\n(6) KERNEL DENSITY ESTIMATES: %s" % png_file
dataRanked = data.iloc[rankedCounties]
dataNotRanked = data.iloc[notRankedCounties]
#dataRanked['log_pop'].plot(kind='kde')
#kde = dataNotRanked['log_pop'].plot(kind='kde')
#kde.get_figure().savefig(png_file)
#plt.show()

#B.7) PROBABILITY RANKED GIVEN POP: Three probabilities
#     --The estimated probability that an unseen county would be ranked,
#     given the following (non-logged) populations: 300, 3100, 5000.
#print "\n(7) PROBABILITY RANKED GIVEN POP:"
def probGivenPop (RPOP_data, NRPOP_data, log_pop):
    """#Approach: The two sets of data (ranked counties and not ranked) can be modeled as norma
       while the question is asking about something being true (i.e. a Bernoulli): P(ranked | p
       where pop is population of the given county
       P(RPOP = pop) = 0 and P(RPOP = pop) = 0, since they are CRVs, but we can simply add an i
       in order to get non-zero values.
       Then, we can look at the ratio of P(pop -.5 < RPOP < pop + .5) to P(pop -.5 < NRPOP < po
       and normalize by the total count of each. (we will work with everything logged)"""

    #first we compute the probabilities under each population
    bw = .5 #bandwidth for estimates
    P_RPOP = ss.norm.cdf(log_pop+bw, RPOP_data.mean(), RPOP_data.std()) - \
        ss.norm.cdf(log_pop-bw, RPOP_data.mean(), RPOP_data.std())#probability among ranked
    P_NRPOP = ss.norm.cdf(log_pop+bw, NRPOP_data.mean(), NRPOP_data.std()) - \
        ss.norm.cdf(log_pop-bw, NRPOP_data.mean(), NRPOP_data.std())#probability among not rank

    #next normalize by population of each to get an estimated number of counties with each popu
    Est_Counties_Ranked_at_pop = P_RPOP * len(RPOP_data)
    Est_Counties_NotRanked_at_pop = P_NRPOP * len(NRPOP_data)

    #finally compute the probability: counties ranked / all counties (in the given population)
    return Est_Counties_Ranked_at_pop / (Est_Counties_Ranked_at_pop + Est_Counties_NotRanked_at_

#print "\t 300: %.4f" % probGivenPop(dataRanked['log_pop'], dataNotRanked['log_pop'], np.log(30
```

```
        #print "\t 3100: %.4f" % probGivenPop(dataRanked['log_pop'], dataNotRanked['log_pop'], np.log(3
        #print "\t 5000: %.4f" % probGivenPop(dataRanked['log_pop'], dataNotRanked['log_pop'], np.log(5


        ## C. Model the health scores as normal. As in (1), consider each column ending in \Value".

        #C.8) LIST MEAN AND STD_DEV PER COLUMN:
        #    For each value column, output it's mean and standard deviation according to MLE,
        #    assuming a normal distribution (pprint a dictionary of {column: (mean, std-dev), ... }).
        #print "\n(8) LIST MEAN AND STD_DEC PER COLUMN:"
        #mean_sd = data[valsColumn].describe()[1:3]
        #mean_sd_dict = dict([
        #    (c, (round(mean_sd[c]['mean'], 4), round(mean_sd[c]['std'], 4), )) for c in mean_sd.columns
        #])
        #from pprint import pprint
        #pprint(mean_sd_dict)
```

```
loading 2015 CHR Analytic Data.csv

(2) TOTAL COUNTIES IN FILE:
                                3141
```

```
In [6]: #since we're about to use numbers in every value column, let's make sure we're dealing with typ
        for c in valsColumn:
            if not (data[c].dtype == np.float64 or data[c].dtype == np.int64):
                data[c] = data[c].apply(lambda val: float(string.replace(str(val),',',''))) ##change to
                data[c] = data[c].astype('float')

        #c.9) PSEUDO-POP-DEPENDENT COLUMNS:
        #    List of columns which appear to be dependent on log_pop.
        #    For the purposes of this assignment, we will call two continuous random variables,
        #    A and B \pseodo-independent" iff  | E(A| B<muB) - E(A| B>muB) | < 0.5*sigmaA.

        print "\n(9) PSEUDO_POP_DEPENDENT COLUMNS:"
        # Let's designate population as variable B, and every other value column as A
        # frist find where B<muB and B>muB:
        dataLt = data[data['log_pop']  < data['log_pop'].mean()]
        dataGt = data[data['log_pop']  > data['log_pop'].mean()]

        #now iterate through each potential A (value column) and test if pseudo-independent
        dep_cols = list()
        for c in valsColumn:
            expected_diff = np.abs(dataLt[c].mean() - dataGt[c].mean()) # | E(A| B<muB) - E(A| B>muB) |
            if expected_diff > 0.5*data[c].std(): #greater than because we're looking for dependent
                dep_cols.append(c)

        print sorted(dep_cols)
```

```
(9) PSEUDO_POP_DEPENDENT COLUMNS:
['2011 population estimate Value', 'Access to exercise opportunities Value', 'Child mortality Value', 'I
```

## 0.1 Hypothesis Testing

### 0.1.1 Hypothesis: The coin is biased.

Formally, $H_0$: A coin is not biased. In other words: $X$ Binomial(n, 0.5)

We want to be 95% sure we can reject $H_0$. We can flip the coin 1,000 and make sure it falls in the middle 95% of a binomial pmf.

In [7]:
```
#We plan to try 1000 tosses:
n_tosses = 1000

#Let's find the middle 95% range:
lower_bound = ss.binom.ppf(0.025, n_tosses, 0.5) ##ppf is inverse cdf (takes percentile, return
upper_bound = ss.binom.ppf(0.975, n_tosses, 0.5)
print "We want a count outside [%d, %d] to reject the null" % (lower_bound, upper_bound)

#now let's flip a coin
def coin_flip():
    return 1 if np.random.rand() > 0.51 else 0 #CHANGE THE COIN BIAS HERE


observed_head_count = np.sum([coin_flip() for _ in range(n_tosses)])#assume we took this from t

print "After %d tosses, we ended up with a count of %d" % (n_tosses, observed_head_count)
if observed_head_count >= lower_bound and observed_head_count <= upper_bound:
        print "Failed to reject the null!"
else:
        print "Null rejected!"
```

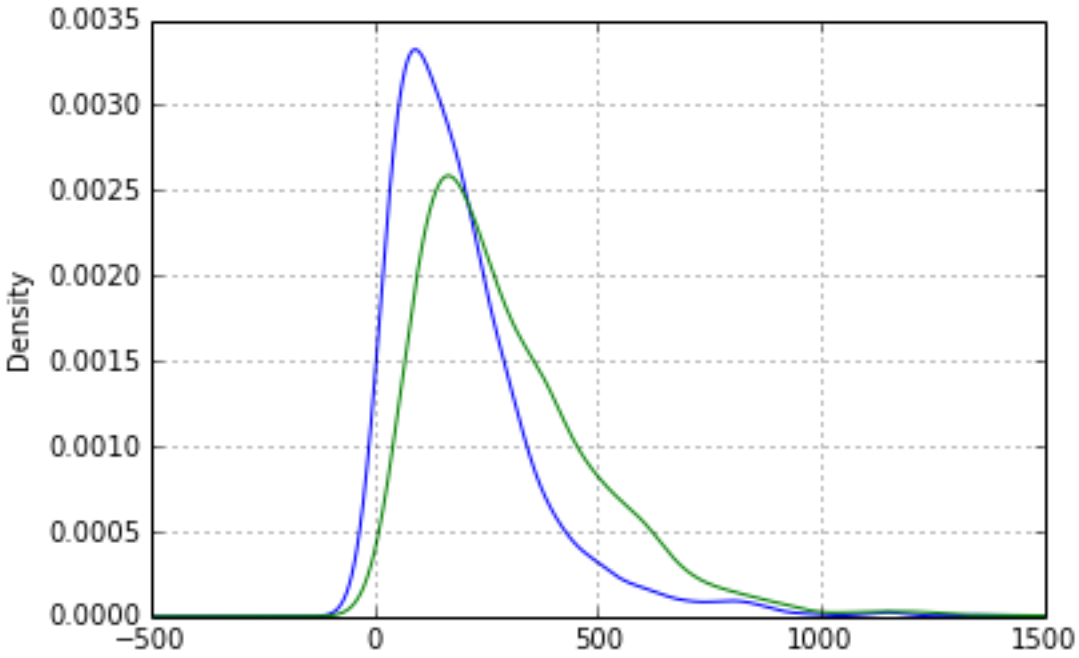
```
We want a count outside [469, 531] to reject the null
After 1000 tosses, we ended up with a count of 504
Failed to reject the null!
```

### 0.1.2 Hypothesis: Communities with higher population have different amounts of violent crimes (per capita) than those with lower population.

In [8]:
```
# Going back to the assignment 1 data:
#dataLt is data for the counties with the lowest half of population
#dataGt is data for greatest population

#let's zoom in on Violent crime and remove nans  (not a number / nulls)
dataLtVC = dataLt['Violent crime Value'][~dataLt['Violent crime Value'].isnull()] #or .dropna()
dataGtVC = dataGt['Violent crime Value'][~dataGt['Violent crime Value'].isnull()]
#dataVC = pd.concat([dataLtVC, dataGtVC])

#let's see as a kde:
dataLtVC.plot(kind='kde')
dataGtVC.plot(kind='kde')
plt.axis([-500,1500,0,0.0035]) #zoom in to these dimensions
plt.show()
```




## 0.2 Using a t-test to compare means

```
In [2]: ##gather means, ns, and standard deviation:
        #mean1, mean2 = dataLtVC.mean(), dataGtVC.mean()
        #sd1, sd2 = dataLtVC.std(), dataGtVC.std() #standard deviation across both
        #n1, n2 = len(dataLtVC), len(dataGtVC)
        #df1, df2 = (n1 - 1), (n2 - 1)
        #print "Mean of lower 50%%: %.1f (%d) \nMean of upper 50%%: %.1f (%d) \n(std across both: %.4f)

        ##two sample t-test, assuming equal variance:
        #pooled_var = (df1*sd1**2 + df2*sd2**2) / (df1 + df2) #pooled variance
        #t = (mean1 - mean2) / np.sqrt(pooled_var * (1.0/n1 + 1.0/n2))
        #p = 1 - ss.t.cdf(np.abs(t), df1+df2)
        #print "t: %.4f, df: %.1f, p: %.5f" % (t, df1+df2, p)
        #print 't-statistic = %6.3f pvalue = %6.4f' %  ss.ttest_ind(dataLtVC, dataGtVC)
```

## 0.3 shape of a student's t distribution

```
In [1]: ##what does a t distribution look like?
        #xpoints = np.linspace(-5, 5, 200)
        #y_3df = ss.t.pdf(xpoints, 3)
        #y_6df = ss.t.pdf(xpoints, 6)
        #y_bigdf = ss.t.pdf(xpoints, 1000000)
        #plt.plot(xpoints, y_3df, linewidth=2, color='#5555ff')#blue
        #plt.plot(xpoints, y_6df, linewidth=2, color='#22cc22')#green
        #plt.plot(xpoints, y_bigdf, linewidth=2, color='#ff5555')#red

        ##how does it compare to a normal?
```

```
#y_znormal = ss.norm.pdf(xpoints, 0, 1)
#plt.plot(xpoints, y_znormal, linewidth=2, color='#dddd33')#yellow
```