Chapter 10; Sampling and the Law of Large Numbers

Steve Elston

01/17/2021

Introduction

Sampling is a fundamental process in the collection and analysis of data. Sampling is important because we can almost never look at the whole population. Further, sampling must be randomized to preclude biases.

Some key points to keep in mind about sampling:
- An understanding of sampling is essential to ensure that analysis performed are representative of the entire population.
- You will use inferences on the sample to say something about the population.
- The sample must be randomly drawn from the population.

Let's look at some examples of sampling.

Use Case	Sample	Population
A/B Testing	The users we show either web sites A or B	All possible users, past present and future
World Cup Soccer	32 teams which qualify in one season	All national teams in past, present and future years
Average height of data science students	Students in a data science class	All students taking data science classes world wide
Tolerances of a manufactured part	Samples taken from production lines	All parts manufactured in the past, present and future
Numbers of a species in a habitat	Population counts from sampled habitats	All possible habitats in the past, present and future

Notice, that in several cases it is not only impractical, but impossible to collect data from the entire population. Hence, we nearly always work with samples, rather than the entire population.

Importance of random sampling

All statistical methods rely on the use of **randomized unbiased samples**. Failure to use such samples violates many of the key assumptions of statistical models. Thus, an understanding of the proper use of sampling methods is essential to performing statistical inference.

Further, most commonly used machine learning algorithms assume that training data are **unbiased** and **independent identically distributed (iid)**. These conditions will only be met if the training data sample is randomized. Otherwise, the training data will be biased and not represent the underlying process distribution.

> **Exercise:** As an example of the problems that can arrise from poor sampling consider the following. A physical retailer wishes to better understand the types of customers in areas where

their stores are located. People are hired to hand coupons to customers entering the stores. The coupons are all worth 5 dollars off a subsequent purchase. The coupons are activated if the customer goes to a web site and spends approximately 20 minutes completing a questionnaire. Coupons are distributed during regular business hours Monday to Friday. There are several sources of biases inherent in this sampling method. Disscuss the sources of these baises, keeping in mind there are several and the scope of your answer is open-ended.

The sampling distributions

We already know that in most practical situations we will only be able to sample from the true data generating process of the population. This sampling is done from an unknown **population distribution**, $\mathcal{F}$. Any statistic we compute for the generating process is actually based on a sample.

We would like to compute a statistic of interest from this distribution of the entire population, $s(\mathcal{F})$. But as we have seen already, we can only compute the statistic from an approximation of this distribution, $\hat{\mathcal{F}}$. Any statistic is actually computed from an estimate, $s(\hat{\mathcal{F}})$.

If we continue to take random samples from the population and compute estimates of a statistic, we say the statistic has a **sampling distribution**. This hypothetical concept is a foundation of **frequentist statistics**. The assumptions of frequentist statistics are built on the idea of being able to randomly resample from the population distribution and recompute a statistic.

In the frequentist world, statistical inferences are performed on the sampling distribution. Random sampling of the population distribution is required so that the statistical estimates are not biased by the sampling process.

Sampling and the law of large numbers

The **law of large numbers** is a theorem that states that **statistics of independent random samples converge to the population values as more samples are used**. We can write this mathematically for the example of the mean as:

$$Let\ \bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Then by the law of Large Numbers:

$$\bar{X} \rightarrow E(X) = \mu as n \rightarrow \infty$$

The law of large numbers is foundational to statistics. We rely on the law of large numbers whenever we work with samples of data. We can safely assume that **larger samples are more representatives of the population we are sampling**. This theorem is the foundation of not only sampling theory, but modern computational methods including, simulation, bootstrap resampling, and Monte Carlo methods. If the real world did not follow this theorem much of statistics, to say nothing of science and technology, would fail badly.

Given this great importance, it should not be surprising that the law of large numbers has a long history. Jacob Bernoulli posthumously published the first proof for the Binomial distribution in 1713. In fact the law of large numbers is sometimes referred to as **Bernoulli's theorem**. A more general proof was published by Poisson in 1837.

Let's think about a simple example. The mean of 50 coin flips (0,1)=(T,H) is usually further away from the true mean of 0.5 than 5,000 coin flips. The code in the cell below computes and then samples a population of 1,000,000 Bernoulli trials with $p = 0.5$. Run this code and examine the results. Does the expected value, or mean converge to 0.5 as `n` increases?

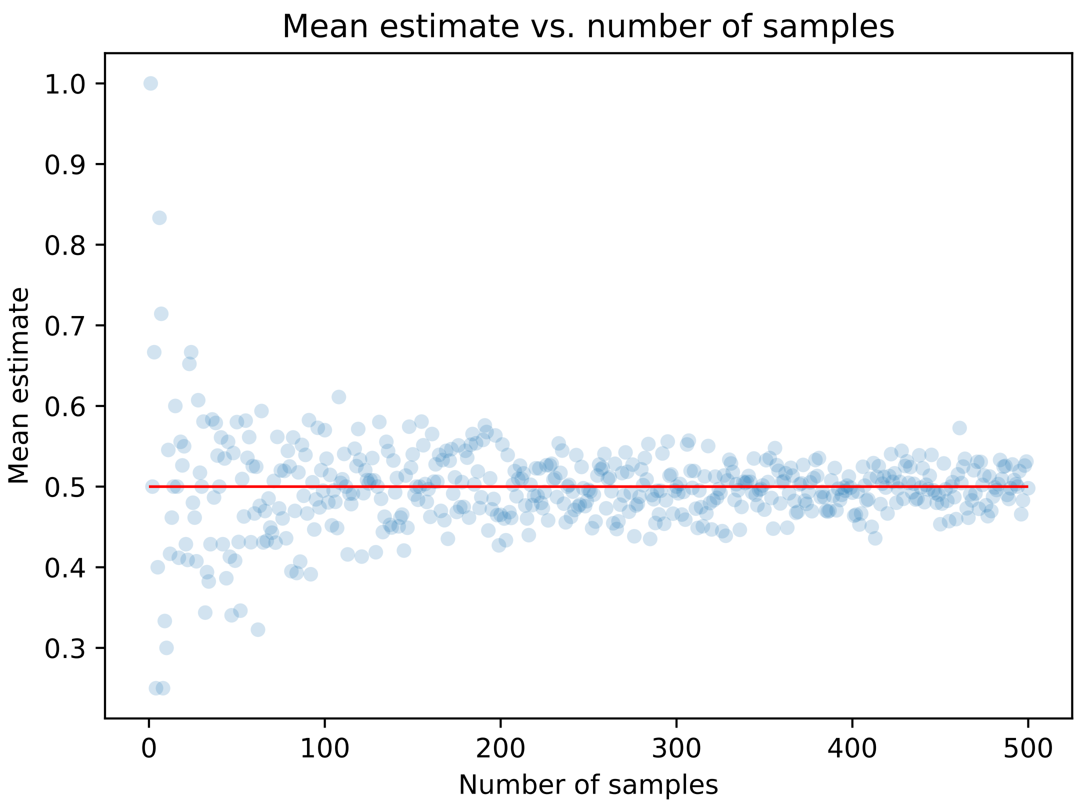
```
nr.seed(3457)
n = 1
p = 0.5
size = 1000000
# Create a large binomial distributed population.
pop = pd.DataFrame({'var':nr.binomial(n, p, size)})
# Sample the population for different sizes and compute the mean
sample_size = [5, 50, 500, 5000]
out = [pop.sample(n = x).mean(axis = 0) for x in sample_size]
for n,x in zip(sample_size,out): print("%.0f  %.2f" %(n, x))
```

```
## 5  0.80
## 50  0.36
## 500  0.48
## 5000  0.49
```

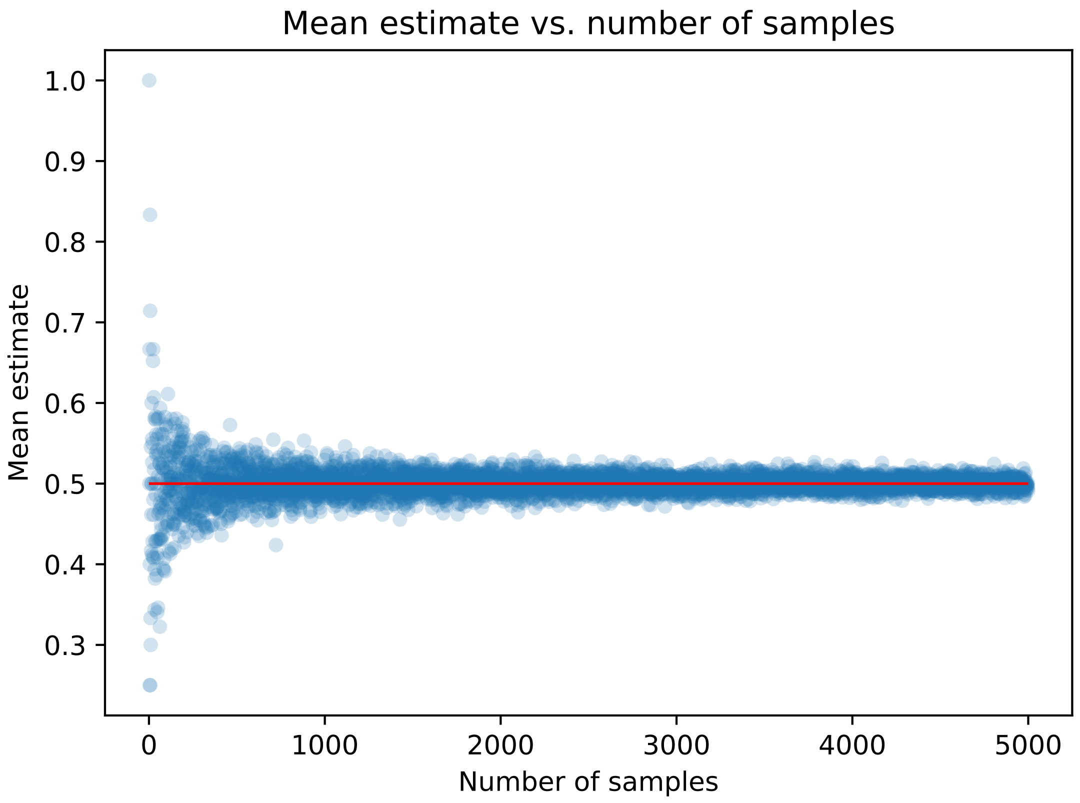
> **Exercise:** You will use the series of Bernoulli trials just computed to examine the convergance of the estimated probability, p. This compuational experiment is equivelent to determing if a long series of flips of a fair coin converge to $p = 0.5$. Now do the following:
> 1. Create and execute code to compute the running mean of the realizations of a series of Bernoulli trials as the sample size increases from 1 to 5000. Your result should be in the form of a data frame with one column containg the number of samples and the other the running mean.
> 2. Create a function to scatter plot the estimate of p vs. the running number of samples. Include a distinctive horrizontal line at the theoretical value of p. Make sure your plot is correctly labled. Run your funtion twice; once for 1 to 500 realizations, and once for 1 to 5000 realizations. 3. Do the estimates of p appear to converge to the known value and the number of realizations increases?

```
# Calculate and plot a running average of N-trials of flipping a fair coin
num_samps = 5000
running_average = pd.DataFrame({'num_samples': range(1, num_samps),
                                'mean':np.concatenate([np.mean(pop.sample(n = x)) for x in range(1, num_
```

```
def plot_samples(df, num_plot, p):
    running_average.iloc[:num_plot, :].plot.scatter(x = 'num_samples', y = 'mean', alpha = 0.2, style="
    plt.hlines(p, 0, num_plot, color='red', linewidths=1.0)
    plt.ylabel('Mean estimate')
    plt.xlabel('Number of samples')
    plt.title('Mean estimate vs. number of samples')
    plt.show()
plot_samples(running_average, 500, p)
```



```
plot_samples(running_average, 5000, p)
```



Standard error and convergance for a Normal distribution

As we sample from a Normal distribution, the mean of the sample will converge to the population mean and the sample standard deviation will converge to the population standard deviation as the number of samples increases. This behavior is expected from the law of large numbers.

But, what can we say about the expected error of the mean estimate as the number of samples increases? This measure is known as the **standard error** of the sample mean. As a corollary of the law of large numbers the standard error is defined:

$$se = \pm \frac{sd}{\sqrt{(n)}}$$

This relationship has significant implications for sampling. The standard error decreases as the square root of n. For example, if you wish to halve the error, you will need to sample four times as many values.

For the mean estimate, μ, we can then define the uncertainty in terms of **confidence intervals**. Confidence intervals are discussed in depth in Chapter XXXX. For Normally distributed values the 95% confidence interval is:

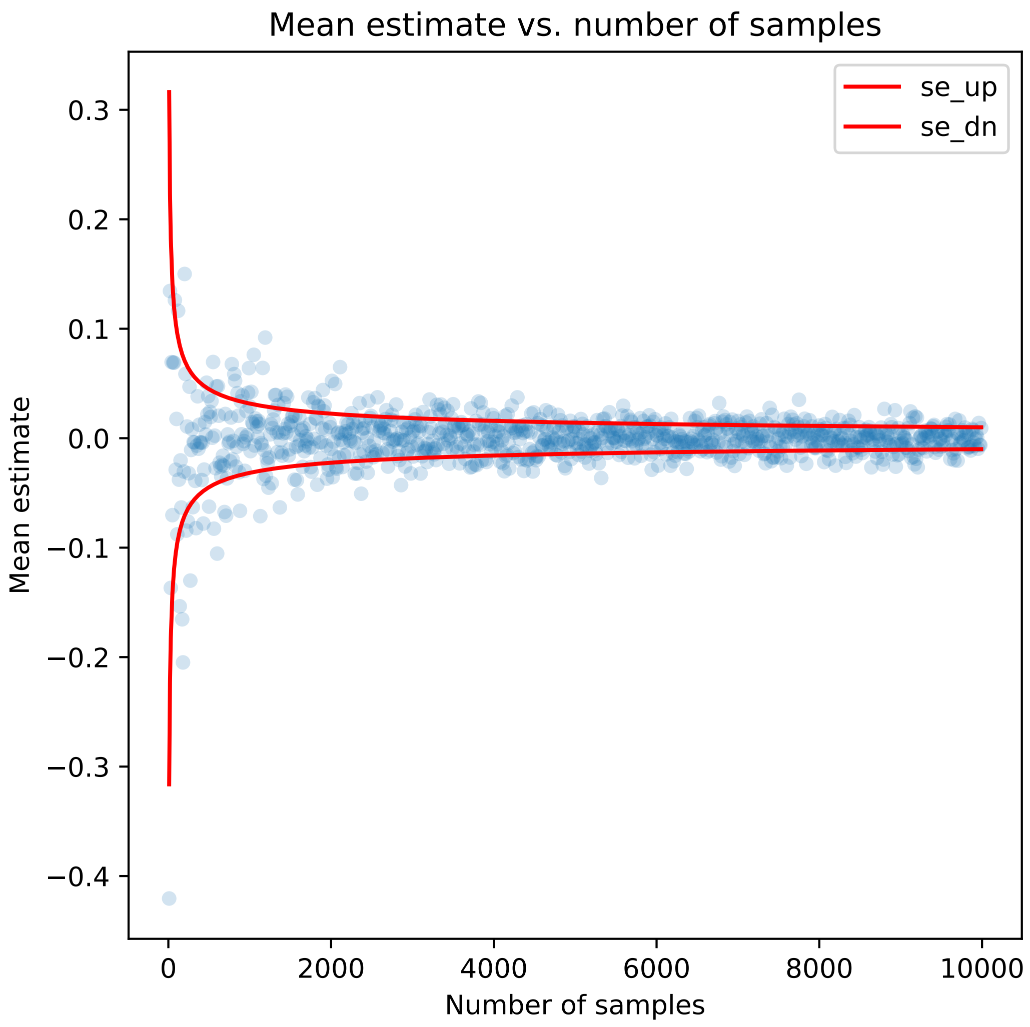
$$CI_{95} = \mu \pm 1.96\ se$$

Exercise: To explore these concepts, you will do the following: 1. Compute the mean using 1 to 10000 samples of from a standard Normal distributon; $\mathcal{N}(\infty, \prime)$.
2. Create a data frame with the following columns; i) the number of samples, ii) the running

mean estimates, iii) the theoretical upper confidence interval, iv) the running estiamted upper confidence interval, v) the theoretical lower confidence interval, iv) the running estiamted lower confidence interval.
3. Create a scatter plot of the mean estiamtes. Include on this plot distinctive lines showing the upper and lower theoretical confidence intervals and the population (theoretical) mean. Include a legend and other required annotation on your plot.

```
## Plot means and SE for 10000 Normal distributions
start = 10
end = 10000
step = 10
norms = pd.DataFrame({'num_samples':range(start, end, step),
                      'mean': [np.mean(nr.normal(size = x)) for x in range(start, end, step)],
                       'se_up':[1/math.sqrt(n) for n in range(start, end, step)],
                       'se_dn':[-1/math.sqrt(n) for n in range(start, end, step)]})
```

```
def plot_se(df):
    ax = plt.figure(figsize=(6, 6)).gca() # define axis
    norms.plot.scatter(x = 'num_samples', y = 'mean', alpha = 0.2, style="o", ax = ax)
    norms.plot(x = 'num_samples', y = 'se_up', c = 'red', ax = ax)
    norms.plot(x = 'num_samples', y = 'se_dn', c = 'red', ax = ax)
    ax.set_ylabel('Mean estimate')
    ax.set_xlabel('Number of samples')
    ax.set_title('Mean estimate vs. number of samples')
    plt.show()
plot_se(norms)
```



Does your plot show convergence to the population mean as you would expect from the law of large numbers?

Exercise: Now that you have a feel for how mean converges following the law of large numbers. The next question to ask is if the variance and therefore the standard error also converge as the number of samples increase. To find out create and execute code to make two plots, one above the other. On the upper, create a scatter plot of the upper confidence interval bound estimates. Include a distinctive line for the theoretical upper confidnce interval. Make sure your plot includes proper annotations. Create the equivelent plot for the lower confidence interval bound.
Now, examine this result. Do the estimates of the upper and lower confidence intervals appear to converge as expected as the number of samples increase?

Sampling Strategies

There are a great number of possible sampling methods. In this lesson, we will focus on some of the most commonly used methods. At the conclusion of this lesson, you should be able to apply (or avoid) these sampling strategies:

- **Bernoulli sampling**, a foundation of random sampling.
- **Stratified sampling**, when groups with different characteristics must be sampled.
- **Cluster sampling**, to reduce cost of sampling.
- **Systematic sampling and convenience sampling**, a slippery slope.

Bernoulli Sampling

Bernoulli sampling is a widely used foundational random sampling strategy. Bernoulli sampling has the following properties:

- A **single random sample** of the population is created.

- A particular value in the population is selected based on the outcome of a Bernoulli trial with fixed probability of success, p.

- The sample has a **fixed or predetermined size**.

Consider an example of a company that sells a product by weight. The company wants to ensure the quality of a packaging process so that few packages are underweight. But, it is impractical to empty and weight the contents of every package. Instead, Bernoulli randomly sampled packages from the production line are emptied and the contents weighed. Statistical inferences are then made based on the sample to determine if the packaging process meets the required standards.

Let's try an example with some synthetic data. The code below creates a data frame with 2000 random samples from the standard Normal distribution. These realizations are generated with numpy.random.normal. Next, these realizations are randomly divided into 4 groups using numpy.random.choice. Notice that the probability of a sample being in one of the groups is not uniform, and sums to 1.0.

```
nr.seed(345)
population_size = 10000
data = pd.DataFrame({"var":nr.normal(size = population_size),
                     "group":nr.choice(range(4), size= population_size, p = [0.1,0.3,0.4,0.2])})
data.head(10)
```

```
##         var  group
## 0  1.469248      1
## 1 -1.150144      2
## 2  2.519226      2
## 3 -0.082478      2
## 4 -0.033601      0
## 5 -1.636656      0
## 6 -0.412092      2
## 7  1.784949      1
## 8  0.042383      2
## 9 -0.619732      2
```

You know that the population of 2000 values was sampled from the standard Normal distribution. Therefore, the mean of each group should be close to 0.0. To find out, the code below does the following: 1. The sample is divided between 4 groups.
2. Summary statistics are computed. 3. A data frame is created with one column containing the count of the numbers, the mean, standard error and confidence intervals for each group.

```
def count_mean(dat):
    import numpy as np
    import pandas as pd
    groups = dat.groupby('group') # Create the groups
    n_samples = groups.size()
    se = np.sqrt(np.divide(groups.aggregate(np.var).loc[:, 'var'], n_samples))
    means = groups.aggregate(np.mean).loc[:, 'var']
    ## Create a data frame with the counts and the means of the groups
    return pd.DataFrame({'Count': n_samples,
                        'Mean': means,
                        'SE': se,
                        'Upper_CI': np.add(means, 1.96 * se),
                        'Lower_CI': np.add(means, -1.96 * se)})
count_mean(data)
```

```
##        Count      Mean        SE  Upper_CI  Lower_CI
## group
## 0       1022 -0.030849  0.032249  0.032360 -0.094058
## 1       2979 -0.000648  0.017912  0.034459 -0.035755
## 2       3994  0.003840  0.015827  0.034861 -0.027181
## 3       2005 -0.014848  0.022617  0.029480 -0.059177
```

Indeed, the means for each group all all close to, but not exactly, 0.0.

> **Exercise:** The foregoing mean estimates look reasonable for the full population of 20000. But, what happens when the population is sampled and the mean is computed from the sample? To find out, do the following: 1. Create a Bernoulli sample of the population with sampling probability, $p = 0.04$. You can use numpy.random.choice for the sampling.
> 2. Compute the mean by group for the Bernoulli sample.

```
p = 0.04
nr.seed(562)
bernoulli = data.iloc[nr.choice(range(population_size), size = int(p * population_size)), :]
count_mean(bernoulli)
```

```
##        Count      Mean        SE  Upper_CI  Lower_CI
## group
## 0         30 -0.165930  0.217143  0.259669 -0.591530
## 1        131 -0.157927  0.086140  0.010907 -0.326760
## 2        166  0.011137  0.078163  0.164336 -0.142062
## 3         73 -0.116601  0.130515  0.139208 -0.372411
```

> Examine these results and answer these questions: 1. Given the number of samples per group do the standard errors and confidencne intervals seem consistent with the number of samples in each group.
> 2. Do the numbers of samples appear representative of the population?
> 3. Does Bernoulli sampling seem optimal when data falls into distinct groups, and why?

Stratified Sampling

You might well ask, why are we concerned with sampling grouped data. Group data is quite common in applications. Just a few examples include:
1. Pooling opinion by county and income group, where income groups and counties have significant differences in population.
2. Testing a drug which may have different effectiveness by sex and ethnic group.
3. Spectral characteristics of stars by type.

What is a better strategy for sampling grouped or stratified data? **Stratified sampling** strategies are used when data are organized in **strata**. The idea is simple: independently sample each equal numbers of cases from each group. The simplest version of stratified sampling creates an **equal-size Bernoulli sample** from each strata.

In many cases, you may need to create nested samples. For example, a top level sample can be grouped by zip code, a geographic strata. Within each zip code, people are then sampled by income bracket strata. Regardless of the exact structure of the hierarchy equal sized Bernoulli samples are collected at the lowest level.

The code below samples our population uniformly, by group. The mean, standard error and confidence intervals are then computed for each group.

```
p = 0.01
def stratify(dat, p):
    groups = dat.groupby('group') # Create the groups
    nums = min(groups.size()) # Find the size of the smallest group
    num = int(p * dat.shape[0]) # Compute the desired number of samples per group
    if num <= nums:
        ## If sufficient group size, sample each group.
        ## We drop the unneeded index level and return,
        ## which leaves a data frame with just the original row index.
        return groups.apply(lambda x: x.sample(n=num)).droplevel('group')
    else: # Oops. p is to large and our groups cannot accommodate the choice of p.
        pmax = nums / dat.shape[0]
        print('The maximum value of p = ' + str(pmax))
stratified = stratify(data, p)
count_mean(stratified)
```

```
##        Count      Mean        SE  Upper_CI  Lower_CI
## group
## 0        100  0.076405  0.107181  0.286480 -0.133670
## 1        100 -0.031878  0.118262  0.199915 -0.263671
## 2        100  0.050156  0.097784  0.241812 -0.141500
## 3        100 -0.086572  0.106724  0.122606 -0.295751
```

> **Exercise:** Answer the following quesitons;
> 1. Are the number of samples identical for each group?
> 2. Are the standard errors and confidence intervals similar of each group?
> 3. How do these standard errors and confidence intervals compare those for the groups with non-stratified samples?

Cluster Sampling

When sampling is expensive a strategy is required to reduce the cost, yet still keep randomization Some examples of data which is expensive to collect includes:

- Surveys of customers at a chain of stores.
- Door to door survey of homeowners.
- Sampling wildlife populations in a dispersed habitat.

In these cases, the population can be divided into randomly selected clusters. The process of cluster sampling follows these steps:

- Define the clusters for the population.
- Randomly select the required number of clusters.
- Sample from the selected clusters. Typically Bernoulli sampelling or some other random sampling is used.
- Optionally, stratify the sample within each cluster.

As an example, you can select a few store locations and Bernoulli sample customers at these locations.

The code in the cell bellow divides a population into 10 clusters.

```
## First compute the clusters
num_clusters = 10
num_vals = 1000
## Create a data frame with randomly sampled cluster numbers
clusters = pd.DataFrame({'group': range(num_clusters)}).sample(n = num_vals, replace = True)
## Add a column to the data frame with Normally distributed values
clusters.loc[:, 'var'] = nr.normal(size = num_vals)
clusters.head(10) # Print the head to get a feel for these data
```

```
##    group       var
## 2      2  0.395300
## 5      5  0.143515
## 9      9  0.944609
## 9      9  0.040266
## 3      3 -0.122846
## 1      1  1.648188
## 4      4  0.580927
## 6      6  0.292871
## 2      2 -0.276425
## 6      6  1.310220
```

> **Exercise:** With these clusters defined you will now perform some sampling to gain a bit of understanding of these methods. Perform the following steps:
> 1. Create and exectue code to find the count of samples in each of the clusters. 2. Randomly sample 3 of the 10 clusters using numpy.random.choice. **Note:** if you do not set a seed, the clusters selected may be different each time your run your code. 3. Bernoulli sample each of the clusters selected with `p = 0.2`, and then stratify these samples by group. 4. Compute and display the mean, standard error and confidence intervals of the strata for each of the three clusters.

```
count_mean(clusters)
```

```
##        Count      Mean        SE  Upper_CI  Lower_CI
## group
## 0         96 -0.280741  0.104011 -0.076878 -0.484603
## 1          91  0.176071  0.101378  0.374773 -0.022630
## 2         106  0.064304  0.096473  0.253392 -0.124783
## 3         104 -0.110558  0.102172  0.089699 -0.310815
## 4          94  0.033989  0.097278  0.224654 -0.156676
## 5          97 -0.052760  0.100585  0.144388 -0.249907
## 6         116 -0.064972  0.095728  0.122655 -0.252598
## 7         107  0.053974  0.078998  0.208809 -0.100861
## 8          87  0.217550  0.104806  0.422969  0.012130
## 9         102  0.081457  0.094341  0.266366 -0.103453
```

```
## Randomly sample the group numbers, making sure we sample from
## unique values of the group numbers.
clusters_samples = nr.choice(clusters.loc[:, 'group'].unique(),
                             size = 3, replace = False)
## Now sample all rows with the selected cluster numbers
clus_samples = clusters.loc[clusters.loc[:, 'group'].isin(clusters_samples), :]
print('cluster sampled are: ')
```

```
## cluster sampled are:
```

```
[print(x) for x in clusters_samples]
```

```
## 9
## 6
## 4
## [None, None, None]
```

```
clus_samples.head(10)
```

```
##    group       var
## 9      9  0.944609
## 9      9  0.040266
## 4      4  0.580927
## 6      6  0.292871
## 6      6  1.310220
## 9      9  0.294119
## 4      4 -2.771963
## 9      9  0.024180
## 6      6 -0.745056
## 6      6  0.070052
```

```
count_mean(clus_samples)
```

```
##        Count      Mean        SE  Upper_CI  Lower_CI
## group
## 4         94  0.033989  0.097278  0.224654 -0.156676
## 6        116 -0.064972  0.095728  0.122655 -0.252598
## 9        102  0.081457  0.094341  0.266366 -0.103453
```

Examine your results and answer these questions:
1. Are the number of stamples in each strata of the clusters equal?
2. Do the mean estiamtes, standard errors and confidence intervals appear representative?

Systematic Sampling

Convenience and systematic sampling are a slippery slope toward biased inferences. These sampling methods lack randomization and the sample created are generally not representative of the population.

As the name implies, convenience sampling selects the cases that are easiest to obtain. A commonly sited example of convenience sampling is known as **database sampling**. For example, the first N rows resulting from a query database query are used for an analysis. There is no reason to believe that such a sample is representative of a population in any way.

Systematic sampling applies some convenient, but not random, sampling method. For example, every k-th case of the population is selected. As you can imagine, this is not a random sampling method, but rather a case of convenience sampling.

> **WARNING:** systematic sampling is a form of convenience sampling. Convenience sampling inevitably leads to incorrect inferences!

Pseudo-Random Number Generation

By now, you likely have an impression that **pseudo-random number generation** is vital to computational statistics. In fact, may areas of computational statistics and machine learning rely on our ability to efficiently and correctly generate these sequences. We will now provide an outline of the basic ideas.

The term, pseudo-random indicates that the algorithms we use do not actually generate truly random numbers, but rather a seemingly random sequence. This pseudo-random sequence is not infinite, but is said to have **cycles**. Good pseudo-random number generators have very long cycles.

A pseudo-random number generator is actually a deterministic algorithm which creates an apparently random sequence of numbers. This sequence continues for the length of a cycle. These algorithms do so by clever sequence of mathematical operations on a set of binary numbers. Long cycles arise from using large prime numbers as multipliers in the generator.

Creation of pseudo-random number generator algorithms has a long and fraught history, predating the computer era. For example, the German Enigma code machines of the Second World War used a set of mechanical rotors to create pseudo-random numbers. The encoded characters of a message were multiplied by the pseudo-random numbers. Decoding the resulting cipher required knowing the seed used to start the pseudo-random sequence. The Bletchley Park code breakers, including Allen Touring, exploited the relatively short cycles of the mechanical pseudo random number generation to decrypt these ciphers.

Even in the mainframe era, pseudo-random number generation suffered from short cycles. The cycles were often unexpectedly. In fact, many of the early algorithms had shorter cycles than was known at the time. This situation resulted in biases from unexpectedly repeating cycles.

It took until the late 20st Century for reliable long-cycle pseudo-random number generator algorithms to be developed. The Mersenne Twister algorithm developed by Matsumoto and Nishimura (1997) is now used in virtually all open source statistics and machine learning packages, as well most commercial products. The name comes from a special class of prime numbers known as Mersenne primes.

The numpy.random.RandomState class, which is used widely in Python analytical packages, uses the Mersenne twister algorithm. The methods of this class perform the transformations for the specified probability distribution.

Pseudo-random number generator algorithms create draws from a Uniform distribution, at least within a cycle. To generate draws from other distributions, the uniformly distributed values are transformed using the inverse of the target cumulative probability function.

Since the algorithms are actually deterministic, the sequence of values can be repeated introducing a **seed value**. If you are creating test cases for statistical algorithms that rely on pseudo-random number generation, you must supply a seed for your test. Otherwise, the test results are not reproducible. If no seed is specified 'entropy' is achieved by queries on counters used by the operating system. In this case, the pseudo-random sequence generated will depend on the state of these counters.

A few more thoughts on sampling

There are many practical aspects of sampling. Here are a few key points to keep in mind when performing sampling. The take-away here is to create a random sampling plan in advance and stick to your plan.

- Random sampling is essential to the underlying assumptions of statistical inference.
- Whenever you are planing to sample data, make sure you have a clear sampling plan.
- Know the number of clusters, strata, samples in advance.
- Don't stop sampling when desired result is achieved: e.g. error measure!

Summary

In this lesson you have explored the concepts of sampling and how sampling relates to the law of large numbers. Nearly all data statisticians and data scientists work with is sampled from larger, unknown, populations. It is therefore important to have an understanding of sampling methods and the convergence of statistical estimates.

Specifically in this lesson you have done the following: - Estimated sample statistics converge to the population statistics as the sample size grows. This convergence is referred to as *The Law of Large Numbers.* - Used Bernoulli sampling to generate randomized samples of a population. - Applied stratified sampling to populations with unequal numbers of members. - Used cluster sampling to reduce the number of samples required for cases where data collection is expensive.

Bibliography