Computer Lab 8: Land snow cover (part 2)

Climate Data Analysis, ATS 301, Fall 2018

As before, we start by specifing that we want plots to be displayed inside the Jupyter Notebook, and load the modules we'll need.

In [ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

# Set default figure size
# This is so I don't have to keep specifying figsize later.
plt.rcParams['figure.figsize'] = (10.0, 8.0)

import xarray as xr
import cartopy.crs as ccrs
from scipy import stats
import math
```

Read in data, exclude ocean points, and points that never have snow.

In [ ]:
```
snow_file='/data/ATS_301/Data/nhsce_v01r01_19661004_20181001.nc'
dsnow = xr.open_dataset(snow_file)
snow = dsnow.snow_cover_extent.where(dsnow.land > 0)
new_snow = snow.where(np.any(snow > 0,axis=0))
```

The reference for this dataset is:

Robinson, David A., Estilow, Thomas W., and NOAA CDR Program (2012):NOAA Climate Date Record (CDR) of Northern Hemisphere (NH) Snow Cover Extent (SCE), Version 1. NOAA National Climatic Data Center. doi:10.7289/V5N014G9 [accessed 10/19/2018].

Verify that this looks like what we expect:

In [ ]:
```
time_average_snow_new=new_snow.mean(dim='time')

ax = plt.axes(projection=ccrs.Orthographic(0, 90))

# This needs to be before the plot function, so that it is at the bottom of the fi
gure.
ax.stock_img()
time_average_snow_new.plot(x='longitude',y='latitude',ax=ax,\
                           transform=ccrs.PlateCarree(),cmap='Greys_r')
plt.show()
```

Grid-point statistics

Our goal here is to determine whether two points have the same mean, standard deviation, etc.

Take a quick look at the time-average plot to identify some particular grid points to explore.

In [ ]:
```
time_average_snow_new.plot()
```

In [ ]:
```
# Compare with the plot with all land points, just to see the effect of omitting
# the snow-free points.
time_average_snow=snow.mean(dim='time')

time_average_snow.plot()
```

Pick a few points. Choose two near each other, and one further away with a different time average. Points with averages near the middle of the colorbar will work best, since they are will have more variation.

Calculate the annual mean for this single point.

In [ ]:
```
# Your indexes will differ, based on the first point you picked.
snow_1=snow[:,47,30].groupby('time.year').mean()
print(snow_1)
```

In [ ]:
```
help(stats.describe)
```

In [ ]:
```
stats.describe(snow_1)
```

Save these values, for later use.

`stats.describe` doesn't include the standard deviation and standard error of the mean, so we use different functions for these.

In [ ]:
```
n1, (min1, max1), m1, v1, s1, k1 = stats.describe(snow_1)

sd1=stats.tstd(snow_1)     # standard deviation
print(sd1)

sem1=stats.sem(snow_1)   # Standard error
print(sem1)
```

Next, calculate the mean, variance, std dev, and standard error yourself, using only `np.sum`, `np.size`, `math.sqrt`, and math operations. Verify that they match what python gets.

Note, you may have to use the `float` function (e.g., `float(mean1)`) to get your values to print correctly.

In [ ]:
```
## Now, calculate the mean, variance, and std dev yourself
mean1= XXXXX
var1= XXXXXX
st_dev1= XXXXXX
st_err1=XXXXXXX
```

In [ ]:
```
# ↔
```

Plot the histogram. You may want to play around with the number of bins it uses.

In [ ]:
```
plt.hist(snow_1, density=True,bins='auto')
# bins='auto' or bins=14
```

Now, what would a normal distribution with this mean and std dev look like?

The `stats.norm.rvs` function samples `size` times from a normal distribution with mean `loc` and standard deviation `scale`.

In [ ]:
```
r = stats.norm.rvs(loc=m1,scale=sd1,size=n1)
print(stats.describe(r))

# Plot our data and the sampled normal data.
plt.hist((snow_1,r), density=True, bins='auto',color=('b','g'))

plt.show()  # used here so that we don't get all the histogram output.
```

Re-run the above cell a few times. Notice how much the distribution can vary.

This is an indication that the limited number of years we have will increase the uncertainty in our statistics, as we will see soon.

Do two distributions have the same means?

Does your grid point data have the same mean as the normal distribution?

They should, since we are specifying the mean value for the distribution. The question is, can we prove it? Or rather, can we disprove that they have the same means?

[Some code is repeated here so it is all in one place.]

In [ ]:
```
# Calculate the annual avereages for each year for this single point.
snow_1=snow[:,47,30].groupby('time.year').mean()

# Statistics for this point.
n1, (min1, max1), m1, v1, s1, k1 = stats.describe(snow_1)
sd1=stats.tstd(snow_1)    # standard deviation (unbiased)
sem1=stats.sem(snow_1)    # Standard error

# Samples from a normal distribution using same mean and std dev
r = stats.norm.rvs(loc=m1,scale=sd1,size=n1)

norm_n, (norm_min, norm_max), norm_m, norm_v, norm_s, norm_k = stats.describe(r)
norm_std=stats.tstd(r)
norm_sem=stats.sem(r)

# Create a table comparing the two

print('                 Snow      Normal')
print('Mean:        {0:.3f}      {1:.3f}'.format(m1, norm_m))
print('Var:         {0:.4f}     {1:.4f}'.format(v1, norm_v))
print('Std:         {0:.3f}      {1:.3f}'.format(sd1, norm_std))
print('SEM:         {0:.3f}      {1:.3f}'.format(sem1, norm_sem))

print('Skew:        {0:.2f}       {1:.2f}'.format(s1, norm_s))
print('Kurt:        {0:.2f}       {1:.2f}'.format(k1, norm_k))


# Plot our data and the sampled normal data.
plt.hist((snow_1,r), density=True, bins='auto',color=('b','g'))
plt.show()  # used here so that we don't get all the histogram output.
```

In [ ]:
```
#↔
```

The **means** and **standard** errors give us an idea of the significance of the difference of the means. More formally, we can do statistical tests to check this.

Here, we're just adding a t-test to the previous code cell.

In [ ]:
```
# Calculate the annual averages for each year for this single point.
snow_1=snow[:,47,30].groupby('time.year').mean()

# Statistics for this point.
n1, (min1, max1), m1, v1, s1, k1 = stats.describe(snow_1)
sd1=stats.tstd(snow_1)    # standard deviation (unbiased)
sem1=stats.sem(snow_1)    # Standard error

# Samples from a normal distribution using same mean and std dev
r = stats.norm.rvs(loc=m1,scale=sd1,size=n1)

norm_n, (norm_min, norm_max), norm_m, norm_v, norm_s, norm_k = stats.describe(r)
norm_std=stats.tstd(r)
norm_sem=stats.sem(r)

# Create a table comparing the two

print('           Snow     Normal')
print('Mean:      {0:.3f}     {1:.3f}'.format(m1, norm_m))
print('Var:       {0:.4f}    {1:.4f}'.format(v1, norm_v))
print('Std:       {0:.3f}     {1:.3f}'.format(sd1, norm_std))
print('SEM:       {0:.3f}     {1:.3f}'.format(sem1, norm_sem))

print('Skew:      {0:.2f}      {1:.2f}'.format(s1, norm_s))
print('Kurt:      {0:.2f}      {1:.2f}'.format(k1, norm_k))


# Plot our data and the sampled normal data.
plt.hist((snow_1,r), density=True, bins='auto',color=('b','g'))
plt.show()  # used here so that we don't get all the histogram output.

# Student's t-test

print()
print(stats.ttest_ind(snow_1,r,equal_var=False))
```

Small p-values indicate that the difference in means is unlikely to be larger than what we obtained here due just to chance.

Large p-values indicate that it's not unlikely you'd get the same difference in sample means from the same distribution by change.

Based on the p-values, we cannot reject the null hypothesis that the mean are consistent.

Repeating the comparison using a different sample of the normal distribution

Try running the above cell again. Do the statistics and results of the Student's t-test change? Does the p-value ever indicate a statistically significant (at the 0.05 level) difference? Should it?

For your homework, you will re-sample from the normal distribution a number of times (~20). Record the t-statistics and p-values. It's easiest to do this using a **for loop** (see below). However, you can also repeat the calculation manually, recording the p values and t statistics each time. Put the t-statistics and p-values into a table.

In [ ]:
```
df = n1+n1-2
x = np.linspace(stats.t.ppf(0.01, df),stats.t.ppf(0.99, df), 100)
plt.plot(x, stats.t.pdf(x, df))

# For your homework, you can either use a for loop to generate these values 20 tim
es,
# or run the previous cell 20 times and record the t-statistics and p values.

# Copy this line 20 times and replace with the t-statistics for your data.
plt.axvline(x=-0.3)
plt.axvline(x=0.19541631165695444)
```

Comparing two normal distributions

The data we have is probably not really a normal distribution. What happens if we compare one random normal distribution to another?

By comparing two sets of normal distributions samples (which we know for sure either are or are not sampled from the same normal distribution population), we can get a feel for how well we can expect the student's t-test to work.

In [ ]:
```
# Create two samples from normal distributions and see how well they compare to ea
ch other.

r1 = stats.norm.rvs(loc=m1,scale=sd1,size=n1)
r2 = stats.norm.rvs(loc=m1,scale=sd1,size=n1)

print(stats.describe(r1))
print(stats.describe(r2))

print(stats.ttest_ind(r1,r2,equal_var=False))

plt.hist((r1,r2), density=True, bins='auto', color=('g','b'))
plt.show()
```

Repeat the above. With what sort of frequency do you get a result that is "statistically significant"?

Now, increase the *number* of values in each sample (and re-run the above cell). What happens?

In [ ]:
```
# Create two larger samples from normal distributions and see how well they compar
e to each other.

n_new=100

r1 = stats.norm.rvs(loc=m1,scale=sd1,size=n_new)
r2 = stats.norm.rvs(loc=m1,scale=sd1,size=n_new)

print(stats.describe(r1))
print(stats.describe(r2))

print(stats.ttest_ind(r1,r2,equal_var=False))

plt.hist((r1,r2), density=True, bins='auto', color=('g','b'))
plt.show()
```

Finally, create two *different* normal distributions and see how well they compare to each other.

In [ ]:
```
# Create two _different_ normal distributions and see how well they compare to eac
h other.

r1 = stats.norm.rvs(loc=m1+sd1/2,scale=sd1,size=n1)
r2 = stats.norm.rvs(loc=m1,scale=sd1,size=n1)


print(stats.describe(r1))
print(stats.describe(r2))

print(stats.ttest_ind(r1,r2,equal_var=False))

plt.hist((r1,r2), density=True, bins=20, color=('g','b'))
plt.show()
```

Even though we know they are different, the test doesn't always come back "significant." Increase the number of data points (i.e. "years") until you usually get a "significant" result.

Compare annual means from two grid points

Finally, we are ready to compare snow data at different points.

In [ ]:
```
# Use your own point here.

snow_2=snow[:,47,31].groupby('time.year').mean()

n2, (min2, max2), m2, v2, s2, k2 = stats.describe(snow_2)
sd2=stats.tstd(snow_2)    # standard deviation (unbiased)
sem2=stats.sem(snow_2)    # Standard error

# Create a table comparing the two

print('                  Snow 1     Snow 2')
print('Mean:        {0:.3f}      {1:.3f}'.format(m1, m2))
print('Var:         {0:.4f}     {1:.4f}'.format(v1, v2))
print('Std:         {0:.3f}      {1:.3f}'.format(sd1, sd2))
print('SEM:         {0:.3f}      {1:.3f}'.format(sem1, sem2))
print('Skew:        {0:.2f}       {1:.2f}'.format(s1, s2))
print('Kurt:        {0:.2f}       {1:.2f}'.format(k1, k2))


# Student's t-test

print()
print(stats.ttest_ind(snow_1,snow_2,equal_var=False))
print(stats.ttest_rel(snow_1,snow_2))

plt.hist((snow_1,snow_2), density=True, bins='auto', color=('b', 'g'))
plt.show()
```

For my two points, I can reject the null hypothesis that these point have the same statistics. (You may get a different result, depending on the points you choose.)

In [ ]:
```
# Pick another point and compare to the first.↔
```

For these two points, I can reject the null hypothesis that these point have the same statistics. (You may get a different result, depending on the points you choose.)

In [ ]:
```
# Test points 2 and 3↔
```

The difference in these two means is not statistically significant according to either test. We *cannot* reject the null hypothesis.

Create a nice table summarizing these three points.

In [ ]:
```
# Determine the location for each point

lat1=float(dsnow.latitude[47,30])
lat2=float(dsnow.latitude[47,31])
lat3=float(dsnow.latitude[46,30])

lon1=float(dsnow.longitude[47,30])
lon2=float(dsnow.longitude[47,31])
lon3=float(dsnow.longitude[46,30])
```

In [ ]:
```
# Put everything in a big table.↔
```

In [ ]:
```
# Indicate the three points you used on a plot. This example has 2, add another.

plt.figure(figsize=[8,8])
ax = plt.axes(projection=ccrs.Orthographic(0, 90))
ax.stock_img()
time_average_snow_new.plot.pcolormesh(x='longitude',y='latitude'\
                                            ,ax=ax,transform=ccrs.PlateCarree(),cmap='Gr
eys_r')

# Within the plt.plot command, indicate the longitude, then latitude
plt.plot(-155,64.9,'r.',transform=ccrs.PlateCarree(),markersize=10)

# The North Pole. No land there!
plt.plot(0,90,'g.',transform=ccrs.PlateCarree(),markersize=10)

plt.show()
```

Using Loops (AKA Making the computer do all the work)

Ok, we've calculated the t-statistics and p-values for a few sample distributions. But we'd like to do 20-50.

Here's where a **loop** is useful. One thing that computers are great at is repeating actions. We can repeat commands over and over (and over) by placing the commands we want to repeat inside a **for loop**. Here's an example that prints "Hello World" 10 times:

```
for i in range(10):
    print("Hello, World")
```

The __indentation__ (the spaces in front of the `print` command) is very important here. This is how the command interpreter knows which commands are within the loop!

The variable `i` is an **index**. `range(10)` is a simple way of creating a **list** of 10 values, from 0 to 9 (i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Remember that Python uses an index of 0 for the first value in a list. If you are used to languages that start with a 1, this behavior will probably cause some bugs in your code at some point, so be aware of the different indexing conventions.

The **`for` loop** runs the code inside the loop once for every value in the list. Within the loop, the variable `i` has whatever the current value from the list is. That is, the first time the body of the loop is evaluated, `i` has a value of 0. The second time `i` is 1. `for i in range(10)` is just a way of saying we will evaluate the statements within the loop 10 times.

Try this simple loop. To see how the index (`i`) changes, we'll output during every **iteration** of the loop. *Notice that the two indented lines are repeated, while the last command is not*.

In [ ]:
```
print('before loop')
for i in range(10):
    print("Hello, World")
    print('The current value of i is ',i)
print('after loop')
```

As you wrote this code, you may have noticed that Jupyter Notebook automatically indents forward, but you have to manually delete spaces when you return to the previous indentation level.

You don't have to stick to indicies of 0, 1, 2, etc. You can also use any list (such as a list of solar constants) to loop over. The commands within the for loop will be repeated once for every index in the list.

Note: do not change the value of your index (i.e., `i` above) within the loop. Bad stuff will happen!

In [ ]:
```
print('before loop')
for solar_constant in np.linspace(300,400,11):
    print("Hello, Earth")
    print('The current value of the solar constant is ',solar_constant)
print('after loop')
```

So, rather than running the following cell multiple times, being sure to record the values:

In [ ]:
```
# Calculate the annual averages for each year for this single point.
snow_1=snow[:,47,30].groupby('time.year').mean()

# Statistics for this point.
n1, (min1, max1), m1, v1, s1, k1 = stats.describe(snow_1)
sd1=stats.tstd(snow_1)    # standard deviation (unbiased)

# Samples from a normal distribution using same mean and std dev
r = stats.norm.rvs(loc=m1,scale=sd1,size=n1)

# Student's t-test
print(stats.ttest_ind(snow_1,r,equal_var=False))
```

and then using those values to make a plot:

In [ ]:
```
df = n1+n1-2
x = np.linspace(stats.t.ppf(0.01, df),stats.t.ppf(0.99, df), 100)
plt.plot(x, stats.t.pdf(x, df))

# For your homework, you can either use a for loop to generate these values 20 tim
es,
# or run the previous cell 20 times and record the t-statistics and p values.

# Copy this line 20 times and replace with the t-statistics for your data.
plt.axvline(x=-0.3)
plt.axvline(x=0.19541631165695444)
```

We can do this all in one cell:

In [ ]:
```
df = n1+n1-2
x = np.linspace(stats.t.ppf(0.01, df),stats.t.ppf(0.99, df), 100)
plt.plot(x, stats.t.pdf(x, df))

for i in range(20):
    r = stats.norm.rvs(loc=m1,scale=sd1,size=n1)
    print(stats.ttest_ind(snow_1,r,equal_var=False))
    t_stat, p_val=stats.ttest_ind(snow_1,r,equal_var=False)
    plt.axvline(x=t_stat)
```

In [ ]: