QBUS6840: Tutorial 2 – Working with Time-Series II

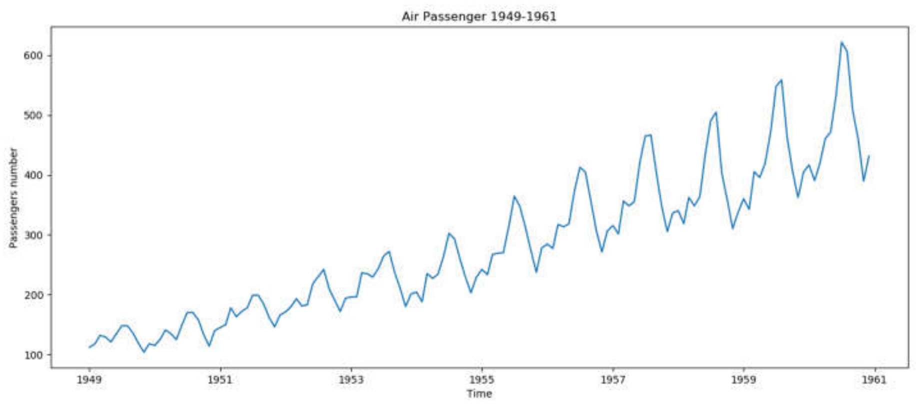
Objectives

- Get familiar with Time Series data manipulation

In our last tutorial, we have learnt the data preprocessing in Time Series, i.e. loading the dataset from a csv file, indexing the data, and visualizing the data. In this week, we will go one step further with data manipulation.

1. Adding noise to your data

Let's continue with step 6 in the last tutorial. At the end of the task, you may get a plot looks like this:



You may notice that the above plot is neat and clean. However, sometimes, this kind of data may not good for the use of time series forecasting due to the lack of generalization. In order to overcome this issue, many people prefer to adding some random noise to the dataset as a form of **regularization**, and therefore, improving the **generalization** of the model.

Here, we will add some noise to our original data. By doing this, we will continue use `numpy` library to add some random noise to the air passenger dataset.

In general, when you have used random numbers in your program, every time when you run your program, it will give different results, causing trouble to validate the problem. To make sure you can obtain the "same" results for all runs, you can start by "seeding" `numpy`'s random number generator (RNG). Seeding can guarantee that each time you run the script the same random data will be produced. You can use any number as the parameter to the seed function. I've chosen 0.

```
np.random.seed(0)
```

Then generate some random data that is the same length as the *Air*

Passengers data:

```
random_data = np.random.randn(len(ts))
```

Then add the following code to plot the data:

```
ts_noise = ts + (random_data * 20)
plt.plot(ts_noise)
plt.title('Air Passenger 1949-1961 number with Noise')
plt.xlabel('Time')
plt.ylabel('Passengers number')
```

Read the above code carefully, and try to answer the questions below:

(1) What is the size of `random_data`?
(2) What is the meaning of "`*20`" and what is the ranging of `random_data`?
(3) If you don't run `np.random.seed(0)` at the very beginning, guess what could happen?

2. Time-shifts

Another common time series-specific operation is shifting of data in time. Pandas has two closely related methods for computing this: `shift()` and `tshift()` In short, the difference between them is that `shift()` shifts the data, while `tshift()` shifts the index. In both cases, the shift is specified in multiples of the frequency.

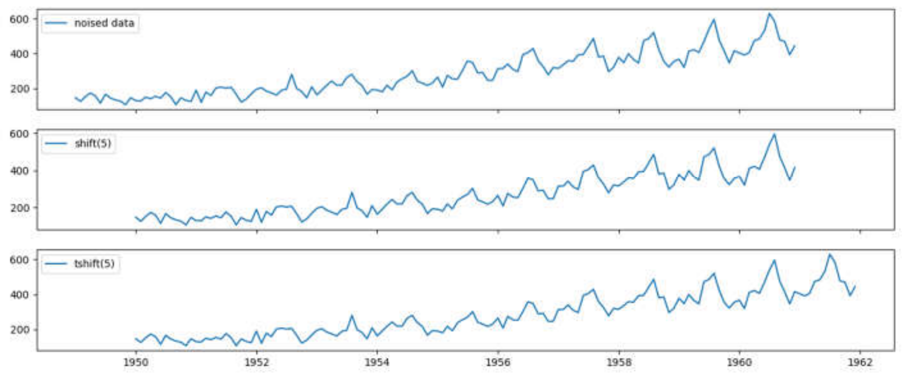
Continue with the last step, here we will use `shift()` and `tshift()` by 12 months:

```
data_shift = ts_noise.shift(12)
data_tshift = ts_noise.tshift(12)
```

Then we plot the original noised data and these 2 shifted data within one window by using `plt.subplots()`. Below is an example,

```
fig, ax = plt.subplots(3, 1, sharex='col', sharey='row')
ax[0].plot(ts_noise)
ax[1].plot(data_shift)
ax[2].plot(data_tshift)
ax[0].legend(['noised data'], loc=2)
ax[1].legend(['shift(5)'], loc=2)
ax[2].legend(['tshift(5)'], loc=2)
```

Run these code, you should see something like this:



Before you go ahead, carefully exam your output and try to answer the following questions:

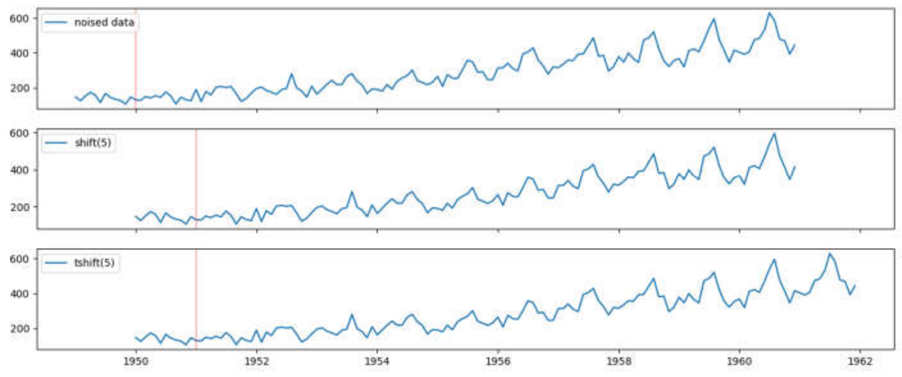
(1) What is the data type of "`ax`"?
(2) What is `ax[0]`, `ax[1]` and `ax[2]`?
(3) Which statement controls the number of rows and columns of the plots?
(4) What are the difference among three new data series?

From the above example, we can see `shift(12)` function could shift the data index with 12 points, which is 12 months in our case. This function pushes some of data point off the end of the graph (and leaving NA values at the other end, please check this from the variable explorer!), while `tshift()` shifts the index values by 12 points and keep do not remove any existing values.

For a better visualization and comparison, you may add some auxiliary lines.

```
local_date = pd.to_datetime('1950-01-01')
offset = pd.Timedelta(12, 'M')

ax[0].axvline(local_date, alpha=0.3, color='red')
ax[1].axvline(local_date + offset, alpha=0.3, color='red')
ax[2].axvline(local_date + offset, alpha=0.3, color='red')
```



3. Retrieving your data

Let's now take a closer look at what we can do with `Pandas` data frames. Download the "drinks.csv" file from the QBUS6840 Canvas site, and use Excel to pre-view the data file so that you understand what information is in the file.

Add the following to your script to import the drinks file:

```
drinks = pd.read_csv('drinks.csv')
```

To view the information about each column of the file add the following:

```
print(drinks.info())
```

Examine the output of the above statement, and answer the following questions

(1) Was the information same as what you have observed in Excel?
(2) What is the data type of the beer servings column?
(3) How many rows do you have? Is the same number of rows in Excel?

We can easily get some basic statistics about each column by adding

```
print(drinks.describe())
```

What is the mean of the wine servings column?

To isolate a single column, you can index a column by its column name. For example, let's isolate the beer servings column:

```
# Extract the beer_servings column as a Series
beer_series = drinks['beer_servings']
# Summarize only the beer_servings Series
print(drinks['beer_servings'].describe())
# Or
print(beer_series.describe())
```

A single column of a `pandas` data frame is a "`Series`" object. `Series` has many of the same features of an entire data frame. Let's calculate the mean of the beer servings:

```
print(beer_series.mean())
```

Pandas allows us to do quite powerful expressions for querying our dataset. For example, you may want to get the countries from Europe. For this dataset try the following:

```
euro_frame = drinks[drinks['continent'] == 'EU']
```

met con1 and con2
df[(condition1) &(condition2)]

met con1 or con2
df[(condition1) |(condition2)]

You can even combine queries. Let’s get all countries from Europe with yearly wine servings greater than 300:

& means "and" | means "or"

```
euro_wine_300_frame = drinks[(drinks['continent'] ==
'EU') & (drinks['wine_servings'] > 300)]
```

Data frames can be sorted. Let’s get the top 10 countries by total liters of alcohol.

```
top_ten_countries =
drinks.sort_values(by='total_litres_of_pure_alcohol').tail(10)
```

Which country drinks most alcohol?

Sometimes your dataset may be missing values. In this case Pandas has interpreted countries from North America (continent = NA) as having “NaN” values. Let’s fix that:

we use string "NA" to replace the null values in the dataframe

```
drinks['continent'].fillna(value='NA', inplace=True)
```

.dropna()

You can insert your own columns into a data frame. For example, let’s create some columns based off the existing columns:

```
drinks['total_servings'] = drinks.beer_servings +
drinks.spirit_servings + drinks.wine_servings
drinks['alcohol_mL'] = drinks.total_litres_of_pure_alcohol * 1000
```

Anyone has spotted “`inplace=True`”in the last step or ask the question about it to your tutor or helper? If you have, congratulations! You are picking up new things.

Finally let’s see if there’s a relationship between beer servings and total litres of pure alcohol.

```
plt.figure()
plt.scatter(drinks['total_litres_of_pure_alcohol'],
drinks['beer_servings'])
plt.xlabel('Total Litres of Alcohol')
plt.ylabel('Beer Servings')
plt.title('International Alcohol Statistics')
```

Summarize what you have seen from the plot!

Summarize what you learnt from this session.