**Experiment 2**

**Name: SAP ID:**

|  |  |
| --- | --- |
| **Date:** |  |
| Aim | **Data Pre-processing: Clean and pre-process a given time series dataset, handling missing values.** |
| Software | Colab |
| Pre-requisite | Internet and required dataset |
| Theory | If you are not already familiar with pandas then you may want to start with our previous tutorial but you should be okay if you understand the concept of a dataframe. It will also help if you are already familiar with the datetime module.  Quantitative work often involves working with time series data in various guises. A time series is an ordered sequence of data that typically represents how some quantity changes over time. Examples of such quantities could be high frequency measurements from a seismometer over a few days, to yearly temperature averages measured at a range of locations across a century, to population changes of different species, but we can use the same software tools to work with them!  In Python it is very popular to use the pandas package to work with time series. It offers a powerful suite of optimised tools that can produce useful analyses in just a few lines of code. A pandas.DataFrame object can contain several quantities, each of which can be extracted as an individual pandas.Series object, and these objects have a number of useful methods specifically for working with time series data.  First import the packages we will use:    1. Loading data into a pandas dataframe  This tutorial will use a heliophysics dataset as an example which contains a range of different measurements. The version of the dataset we will use is available as a raw text file and contains hourly measurements from the beginning of 1963 onwards. This type of file (often .dat, .txt, or .csv) is the least sophisticated and is not the right solution for larger datasets but is okay here (the file is around 150MB) - large/new datasets will often use formats like HDF or NetCDF. pandas contains a range of IO tools for different formats - look here when you want to read or write a dataset.  Link: Download the dataset from <ftp://spdf.gsfc.nasa.gov/pub/data/omni/low_res_omni/omni2_all_years.dat> and take a quick look at the accompanying description: [ftp://spdf.gsfc.nasa.gov/pub/data/omni/low\_res\_omni/omni2.tex](tftp://spdf.gsfc.nasa.gov/pub/data/omni/low_res_omni/omni2.text) Look for OMNI2\_YYYY.DAT FORMAT DESCRIPTION to see the list of columns contained in the dataset.  columns 1, 2, 3 giving the year, day of year (DOY), and hour of day of each measurement  column 40: the sunspot number (R) - the number of spots on the surface of the Sun, indicating how active it is  column 41: the Dst index - an hourly magnetic activity index measured at Earth’s surface, in nT  column 51: the F10.7 index - the radio flux at 10.7cm (i.e. how bright the Sun is at that wavelength), in “solar flux units” (sfu)  We will investigate this data to see if there is a connection between conditions on the Sun (R and Dst), and magnetic conditions at Earth (Dst).        **2. Creating a datetime index**  Now we have the data loaded, we want to fix it a bit to make it more useful. First we will change the index from its current state as a sequence of integers to the more functional pandas.DatetimeIndex which is based on Python datetime objects,.  We use the pandas.to\_datetime() function to create the new index from the “Year”, “DOY”, and “Hour” columns, then assign it directly to the .index property of df, then drop the unneeded columns:    df["Year"] \* 100000 + df["DOY"] \* 100 + df["Hour"] combines the columns into one column of fixed-width numbers following the [YearDOYHour] pattern that can be parsed by the "%Y%j%H" format specifier. df.head() should now show:    When working with other data, you will need to find an appropriate way to build the index from the time stamps in your data, but pandas.to\_datetime() will often help. Now that we are using a DatetimeIndex, we have access to a number of time series-specific functionality within pandas.  In this dataset, data gaps have been infilled with 9’s. We can replace these occurrences with NaN:    This tells us that there is the same number of records in the dataset as the number of hours between the first and last times sampled. We are dealing with over 55 years of hourly samples that results in about half a million records:    **3. Plotting dataframe contents**  The data should now be in an “analysis-ready” format and we should start inspecting it. Let’s start by using the .plot() method. Try each of the following and compare what you get:    Now you can start to get a feel for the data. F10.7 and R look well correlated, each with 5 peaks evenly spaced over time. There is a lot of noise in all the measurements, and it is hard to see any relation with Dst. So what can we do to look deeper for trends and relationships?    **4. Resampling, rolling calculations, and differencing**  To reduce the noise in the data, we can smooth it. There are various ways to do this and so there is a choice to be made about the method to use and the degree of smoothing required. pandas offers a convenient way to reduce the data cadence by resampling with the .resample() method:    Here we have extracted a dataframe with the columns we are interested in with df[["F10.7", "R"]], produced a year-based “resampler” object, which is then reduced to the new yearly time series by taking medians over each year interval.  .resample() has given us a lower cadence dataset which then doesn’t contain the high frequency noise. Similar to this are rolling window calculations, which return the same cadence of data as the input but calculations are performed over a rolling window of a given width about each datapoint. We can use the .rolling() method to do this. Here we construct a moving median filter:    Rolling calculations take the size of the window as the argument, whereas resampling takes a frequency specifier as the argument. NB: we can now see the appearance of some gaps in the F10.7 time series since by default no gaps are allowed within each window calculated - this behaviour can be changed with the min\_periods argument.  Take a look at the documentation to see what other calculations can be done on resampler and rolling objects.  Differencing is often a useful tool which can be part of time series algorithms. See for example how we can use smoothing and differencing to more clearly isolate the periodic signal:    **5. Identifying periodicity and correlation**  We can see by eye that there is an approximately 10 year cycle in R and F10.7. A handy high level tool to identify this periodicity is pandas.plotting.autocorrelation\_plot():    This produces an autocorrelation plot: the correlation of a time series with itself at a range of lag times. We have applied it to the downsampled yearly time series which makes the calculation a lot quicker. Since the cadence of the time series is one year, the “Lag” axis is measured in years. The first peak (after a lag of 0) is around 11 years, meaning that the series correlates well with itself at a lag time of 11 years. This is the well-known solar activity cycle.  Let’s look again at the Dst index and try to find if there is a connection to R. It is helpful to consider the context of the quantities we are examining. R, the sunspot number, indicates solar activity, and Dst indicates geomagnetic activity, the magnetic field created by time-varying large scale electric currents around Earth. We could try to smooth Dst as well to try to reduce the noise to see if there is some correlation with R, but I can tell you now that it will be tricky to prove something from that. Variations in Dst actually tend to occur in discrete events called “geomagnetic storms”, where Dst suddenly drops well below 0nT and takes some hours or days to recover back to 0. We can classify a large storm as when Dst drops below -100nT. Let’s use this to search for occurrences of large storms!  We can mask out where Dst drops below -100 with df["Dst"].where(df["Dst"]<-100), and then count how many entries there are each year that satisfy this condition:    It looks like there is a correlation between high sunspot numbers (the peaks of the solar cycle) and the occurrence rate of large storms. However, there is a lot more variation in this storm rate - lots of sunspots doesn’t guarantee lots of storms, and storms can still occur when there are few sunspots.  **6. Splitting and stacking cycles**  Let’s split the time series up into its constituent cycles and stack them together. This requires some more complex work with pandas and matplotlib. At this point we will also down sample to a daily rate, which makes the plot a bit clearer and quicker to generate.    We now have a list, cycles, containing five dataframes, each containing a different cycle. On each dataframe, we have changed the index into the number of days from the minimum, and used [.reindex()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.reindex.html#pandas.Series.reindex) to fix them all to the same length so that we can perform arithmetic operations on them together. The following will create a plot of each parameter, with the cycles superposed over each other. In this example, we first create the figure and its axes using matplotlib directly (using sharex=True to link the x-axes on each plot), then direct the pandas plotting commands to point them to the axis we want each thing to plot onto using the ax kwarg. We also calculate the mean of the stacked time series.      This helps us to see how the cycles differ from each other: for example, the most recent cycle is consistently lower than the mean, both in the solar conditions and the rate of geomagnetic storms. By constructing the mean of the cycles, we are actually reinforcing the similar pattern over each cycle and reducing the effect of the random noise. This is the basis of a technique called superposed epoch analysis, which is useful for identifying periodicities and similarities between noisy time series. |
| Result |  |
| Conclusion | Write your detail understanding about the experiment. |