BDiF
Big Data in Finance

Baruch "Big Data" course ~ Assignment A

Document name: "BDiF2017 - Baruch Big Data Course - Assignment A"
Date/revision: Fri 10th February 2017 / Rev E.
Author: Andrew Sheppard, © 2016-2017.

Title: "Building a Big Data (parallel) data scrubber and analyzer"
When issued: Fri, 10th February 2017
When due: 12 noon, Fri, 3rd April 2017
Contact details: andrew.sheppard@baruch.cuny.edu

1. Preamble ~ "The Data Hole that Was"

Long, long ago I worked in atmospheric physics at Oxford University. Most people have heard about the "ozone hole" phenomenon; the springtime decrease in stratospheric ozone over Earth's polar regions. Oxford was a pioneer of ozone measurement. When the ozone hole was first discovered, it was thought to be a recent phenomenon. However, when data sets over the 30 years pre-dating satellite data were re-examined, it was found that the hole had been there all the while!

How could this have been missed? Well, it turns out that the scientists when gathering the data had decided to set what they supposed were reasonable upper and lower bounds for data points so as to reject "bad data". The result was that the "ozone hole" was simply ignored for 30 years because the hole data points fell below the lower limit.

This story illustrates the dangers of filtering raw data. You run the risk of missing something important, and it is a cautionary warning as you embark on this assignment.

Filtering, bounds checking, or cleaning data – however you want to put it – is collectively called "scrubbing" data in this course and is more precisely defined like this: scrubbing financial data is the removal of data points that would otherwise cause a human or machine to draw the wrong conclusions. What is particularly important is that this definition leaves open the possibility that the scrubbing doesn't have to be perfect; just "good enough".

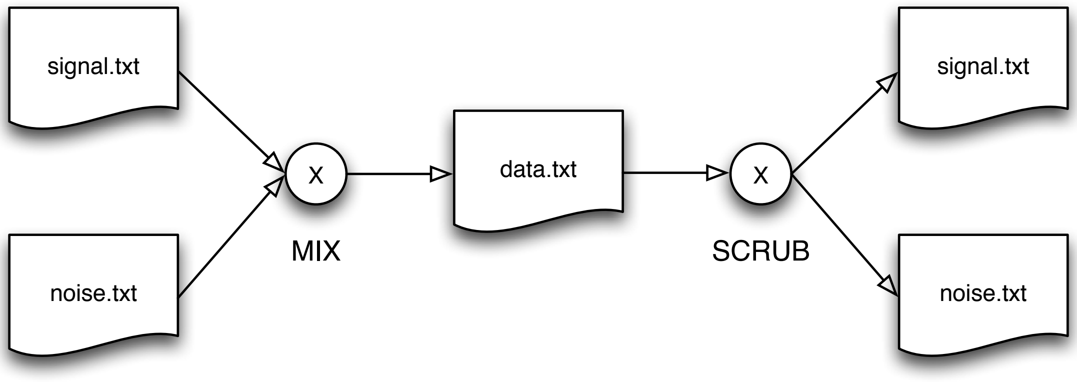
"Good enough" is important for two reasons. First, financial data typically has a short half-life, which combined with a high data rate means that for real-time decision making purposes it must either be scrubbed quickly or, frankly, thrown away. Second, because of the high data rates and real-time requirement, perfection is likely unattainable.

At the same time, in finance, tail events (rare, but severe events) are often the most life threatening (from a P&L and risk point of view), so it is essential that those be absolutely unambiguously captured if real, and rejected equally unambiguously if false.

2. <u>Problem Description</u>

Consider two time series data files, which we will name “signal.txt” and “noise.txt”, and which are generated synthetically, or by hand, or by a mix of the two methods. These two time series are merged together to produce a single file which will be called “data.txt”.

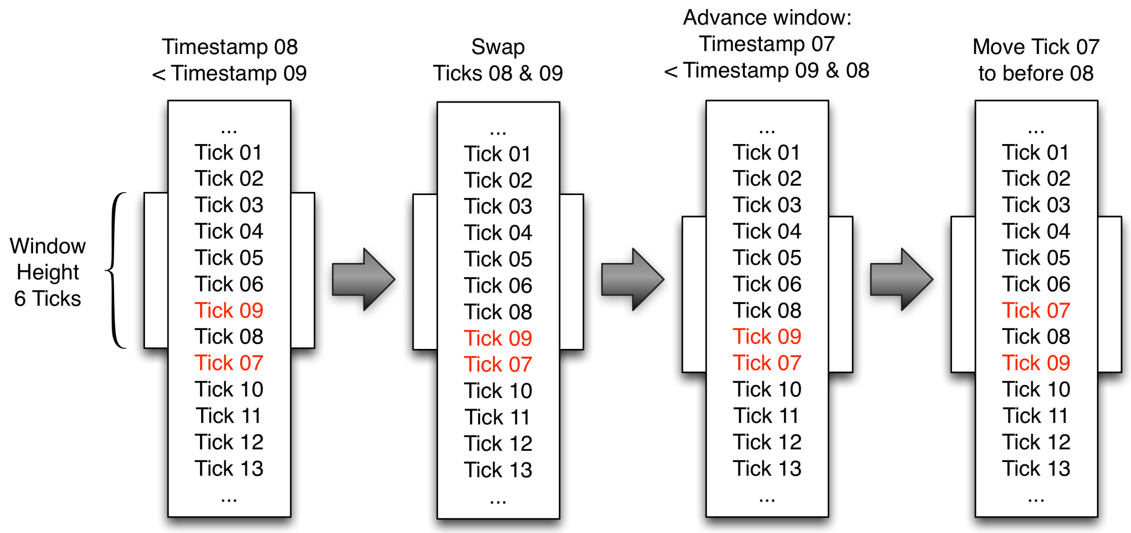
Your task is to take in “data.txt” and output your best guess at “signal.txt” and “noise.txt”, as shown in the figure below.



Your program “SCRUB” must separate the signal from the noise

Note: It's OK, and indeed preferred, if you don't modify "data.txt". It's perfectly OK to create a "noise.txt" file that simply lists the bad ticks and their line numbers from the original "data.txt"; you can then simply say that "signal.txt" = "data.txt" minus "noise.txt" (that is, "signal.txt" is merely a logical and not physical file).

An added wrinkle is that data points in “data.txt” may be out-of-order and so the first thing you will have to do is put them into time order; this can happen, for example, because data packets are routed via different paths over a network. However, this “out of order” property is a localized phenomenon in that ticks are delayed by seconds, or rather more likely fractions of a second, rather than minutes, hours or days. Ticks from three days ago mysteriously appearing on the network and being included into the time series data stream simply doesn’t happen! Also, given an live time series is infinite in extent (or should be considered so) going back into the distant past and stretching far into the future, the sorting and order must always be a localized phenomenon because you never have the whole series. In an infinitely long time series, the concept of global sorting doesn't make much sense, but sorting data points in a time series in a sliding "window" does make sense. What goes into the window is perhaps out-of-order data points and what emerges from the window are in-order data points, but any sorting is localized to that window; which, for any given data point, is chosen to encompass enough linear time to catch any out-of-order data points either side of it, as shown in the figure below.



Sliding window and sorting out-of-order ticks within the window

The scheme shown above is just one way to effectively order data points locally and only works if the window is chosen to have a height greater than the maximum distance between out-of-order ticks (in this case, perhaps we know that out-of-order ticks are never more than 6 ticks apart). As the window moves downwards it leaves data points in time order behind it. Note that the reason for ordering the data within the window is to simplify your analysis, such as detecting duplicate data points, which are adjacent to each other if the ticks are sorted. But, importantly, while the data points are sorted in memory (fast), it does not have to mean that the original data needs to be reordered on disk (slow). The window also need not advance just one data point at a time; it may advance in blocks. And, lastly, you are not necessarily looking for perfection, so "good enough" is OK and having perfect sort ordering at every step of the analysis is maybe at odds with "good enough".

It has been observed that many types of asset returns are distributed normally or, perhaps more accurately, approximately log-normal. But much of that work was done long before machine trading became so important. As this assumption of "normality" underpins much of quantitative finance for many basic models and results, let's check it with real data.

To do this you will write a second program called "NORMAL" that will analyze the scrubbed output of your "SCRUB" program to explore its properties. You will specifically look at the asset returns to see if normality is, or is not, confirmed.

For comparison, you will also pass a long time series of gold prices through your "NORMAL" program and again test for the presence or absence of normality.

Your program "NORMAL" must test time-series for normality, or its absence.

3. <u>**What This Assignment is Meant to Test**</u>

This assignment is meant to test both your data exploration skills and data programming skills; two key components of the Big Data methodology that you are taught on the course (the other two being data analysis and data insights).

Poor data quality is a simple fact of life, particularly for financial data. To do anything useful with such data requires it to be scrubbed, and in the case of financial data for it to be scrubbed quickly. It's one of the uglier sides of Big Data, so we may as well face the data quality demon now.

This assignment will also require you to think creatively about input/output, efficient memory use and data structure structures, scalability, and setting up data for algorithms to be later used downstream.

4. <u>**Problem Specification**</u>

"data.txt" will contain a number of records having the following format:

- Column 1: Timestamp in the format YYYYMMDD:HH:MM:SS.SSSSSS (1 microsecond resolution).
- Column 2: Price as floating point in text form. The "price" in this case will be the price of an unknown asset class.
- Column 3: Units traded as an integer in text form.
- The data file will be in CSV format. That is, columns are separated by commas.

Your program must work with "data.txt" files across a range of sizes, from hundreds of KB up to many GB and even TB! For development purposes you will be given just one fairly large "data.txt", but for final testing your program will be run against a number of "data.txt" files of varying sizes up to the maximum size, so you had better make sure your program scales. The reason for doing things this way is because in the real world you only get to see past data, yet your program must run – and run well – against future data.

At a minimum, your "SCRUB" program must:

- Read in "data.txt" and store it in memory for analysis. This will have to be done in chunks, or as a stream of data.
- Output the cumulative time taken to read the data, parse and to output it.
- Output the time taken just to scrub the data, excluding time for I/O.

- Output your version of "signal.txt". May be a modified version of "data.txt". Does not need to be sorted.
- Output the time taken to write "signal.txt" to disk.
- Output your version of "noise.txt". Does not need to be sorted.
- Output the time taken to write "noise.txt" to disk.
- More generally, your program should output a log file that tracks the performance of your program while it is running and provides the execution times and memory usage of its key parts.
- Describe your program in no more than a single page of text and diagrams, paying particular emphasis on the tradeoffs you have made and anything you thing novel in your approach. This need not be typed or drawn on a computer; a handwritten note, if clear and concise, is fine.
- The program must be written in Python using MPI (http://pythonhosted.org/mpi4py/).
- Your Python code must run on Linux (Penzias cluster).
- The maximum size data set you program will be tested with will require you to attack that file in parallel. In this case you will be writing a distributed parallel program using MPI, and perhaps MPI I/O (your choice on the latter).
- Your program should be invoked at the command line thus (you may have command line parameters or a parameters file):
    - $> python scrub.py data.txt

At a minimum, your "NORMAL" program must:

- Accept time series of arbitrary length in the format described above.
- Generate output that gives the primary characteristics of the data.
- Provides a meaningful measure or the normality/non-normality of the time-series.
- Your program should output a log file that tracks the performance of your program while it is running and provides the execution times and memory usage of its key parts.
- Describe your program in no more than a single page of text and diagrams, paying particular emphasis on the tradeoffs you have made and anything you thing novel in your approach. This need not be typed or drawn on a computer; a handwritten note, if clear and concise, is fine.

Times output must be accurate to 1 ms.

Final tests will be carried out on the CUNY supercomputing cluster. All programs and documentation must be submitted via your Github account.

5. <u>**Some Hints to Help You Along**</u>

Given that you will only have only one (or perhaps a few of varying sizes) sample "data.txt", you may want to synthetically generate files of varying sizes having similar properties for test purposes. (Hint within a hint: if you can't guess or approximate the underlying process that generated the original values, then consider time-reversal, concatenation, inserting of interpolated points, random injection of errors, resampling and a host of other ways to generate synthetic data.) This is useful skill to have as it effectively arbitrarily increases your test data set. And be pragmatic in your use of tools: does this part need to be written in Python, or some other tool such as Awk? It's OK to mix and match tools, although your main scrub.py program should be written in Python.

Double or float? (Actually a trick question for Python. Why?) Maybe it's worth pre-scanning the data set to determine if float will be "good enough" because that would give a 2:1 advantage in memory storage. Likewise for integer and short integer values. Is the effort in pre-scanning worth the memory saving? (Hint within a hint: the danger of writing a program that makes assumptions on only a sample data set is that those assumptions may not hold true for future data sets. One approach is to test data items as they are ingested so that the program can adapt to new data. For example, a short integer that spills to an integer could throw an exception and restart the program from the beginning of the file using a more appropriately sized data structure. What are the disadvantages of this approach?)

For data structures, you may want to consider not just compactness, but data alignment.

As you write your program, start thinking about and building your Python toolbox for Big Data.

Look at real-world asset prices over a long period of time. What important patterns can you see? Any "regime changes" over time?

At no point should you sort the whole "data.txt" input file, though "noise.txt" may be more useful if sorted (but "noise.txt" <<< "data.txt" and "signal.txt"). You may do some in-memory sorting as needed as your program runs, but sorting Big Data files is expensive, so only do it when you have to (and are asked to!).

6. <u>**How This Assignment Will be Graded**</u>

Partial programs that do not meet all the requirements described above will receive credit. You will get marks for showing you were heading in the right direction.

Points will be allocated as follows:

- 20% for meeting the minimum requirements of the program specification for the smallest "data.txt" input file. If your programs output all that is required for the smallest "data.txt" you will have earned 20%; otherwise you will earn some fraction of 20%.

- 20% for programming style. This encompasses not just the clarity of your code, but also such things as efficient data structures, efficient algorithms and use of system APIs and resources. While use of libraries is generally encouraged, there is no substitute for understanding the underlying algorithms, so for this assignment implementing your own algorithms rather than relying completely on library functions will get you more points. However, get your programs working correctly first, and only substitute library functions with your own code if you find yourself with time left over.
- 20% for scalability. If your programs run to completion against all input files of various sizes, you will receive 20%; otherwise you will receive some fraction of 20%.
- SCRUB (Data programming): 20% for the accuracy of your "signal.txt" and "noise.txt" by diff'ing with the original input files. Also, a portion of points in this category will be given for the overall execution time of your program; the shorter the run time the better your mark!
- NORMAL (Data exploration): 20% for the analysis provided by your program in exploring the time-series data processed.

7. **Final Comments**

This is a challenging exercise, so if you find yourself struggling, reach out to me sooner rather than later. And remember that this is not a pass/fail exercise and that you can still get a good grade with a less than perfect program.

As always, doing a little programming each day will get you to a working solution more quickly and with less stress than leaving things until a few days before the submission data. Just a piece of advice.

Important note: This is an individual assignment. The work you submit must be your own, and wholly your own; otherwise you will be penalized.