Message jitter exercise: code explanation & findings

by Trevor Foote

Part 1 of 2: Code explanation

When looking at the data in `logdata.csv`, I quickly realized the magnitude of the quantity of messages sent (I've never seen a 5.19 MB CSV file before); thus I knew run-time could become an issue when sorting / searching through this data. While there are 350,244 lines in `logdata.csv`, there are only 102 unique message IDs. Nevertheless, I thought it would be a good idea to organize the message IDs into a Binary Search Tree data structure; with each object representing a message and containing its message ID, period, and list of timestamps.

If the client wants to search for a particular message ID and pull up its period & timestamps, searching through an unsorted list of message IDs has runtime complexity *O(n)*; while searching through a sorted list (using Python's built-in sorted() function) has a considerably better runtime complexity of *O(log n)*. While it may seem like searching through a sorted list would offer similar runtime performance as searching through a BST (with the added benefit of less coding involved), it turns out there are advantages to the BST data structure: namely in insertion and deletion runtimes.

To implement this data structure, I created a class called `LogTree`, with its attributes seen below:

```
class LogTree:
    _message_id:    Optional[int]
    _period:        Optional[Dict[int, int]]
    _timestamps:    Optional[Dict[int, List[float]]]
    _left:          Optional[LogTree]
    _right:         Optional[LogTree]
```

Each `LogTree` object represents a message and stores that message's ID, period, and a list of timestamps. Additionally, it stores its 2 subtress in `_left` and `_right`. The reason why all attributes are optional is that some `LogTree`s are "empty", and this is denoted by all 5 attributes being `None`. All non-empty `LogTree`s must have `LogTree`s in their `_left` and `_right` attributes; so `LogTree`s without subtrees have empty `LogTree`s as subtrees. This system is necessary to write the recursive methods such as the search method `__contains__` and the string representation method `__str__`.

The class's initializer takes 2 dictionary arguments: the first mapping message IDs to their periods, and the second mapping message IDs to a list of their timestamps; with a pre-condition being that the message IDs in both dictionaries must match:

```
    def __init__(self, period_dict: Dict[int, int] = None, timestamps_dict: \
Dict[int, List[float]] = None):
```

The initializer then finds the median message ID, and that `LogTree` then represents that message. All other messages in the dictionaries are subtrees of this `LogTree`. The initializer then creates 2 sub-dictionaries – one with message IDs less than the median, and one with message IDs greater than the median – and uses recursion until every message has been made into a `LogTree` object and the tree is complete:

```
    if len(period_dict) == 1:
        self._left = LogTree()
        self._right = LogTree()
    elif len(period_dict) == 2:
        self._left = LogTree(
            dict(sorted(period_dict.items())[:floor(len(period_dict) / 2)]),
            dict(sorted(timestamps_dict.items())[:floor(len(period_dict) / 2)])
        )
        self._right = LogTree()
    elif len(period_dict) > 2:
        self._left = LogTree(
            dict(sorted(period_dict.items())[:floor(len(period_dict)/2)]),
            dict(sorted(timestamps_dict.items())[:floor(len(period_dict)/2)])
        )
        self._right = LogTree(
            dict(sorted(period_dict.items())[floor(len(period_dict)/2)+1:]),
            dict(sorted(timestamps_dict.items())[floor(len(period_dict)/2)+1:])
        )
```

There is also an `__str__` method, which allows the client to use built-in python functions such as `print()` and `str()` with `LogTree` objects. For example, when a `LogTree` called `log` is made using data from the 2 CSV files provided to me, `print(log)` returns the following:

This is a visual representation of the Binary Search Tree containing all 100 message IDs. (Note: there are 2 message IDs (321 and 305) that are present in `logdata.csv` but not `periods.csv`, thus they are ignored.)

The `__contains__` method allows the client to use built-in Python features such as the `in` keyword to determine if a message ID is in the tree. This method was also written recursively, using binary search with *O(log n)* time complexity:

```
  def __contains__(self, item) -> bool:
      if self.is_empty():
          return False
      elif item == self._message_id:
          return True
      elif item < self._message_id:
          return item in self._left
      else:
          return item in self._right
```

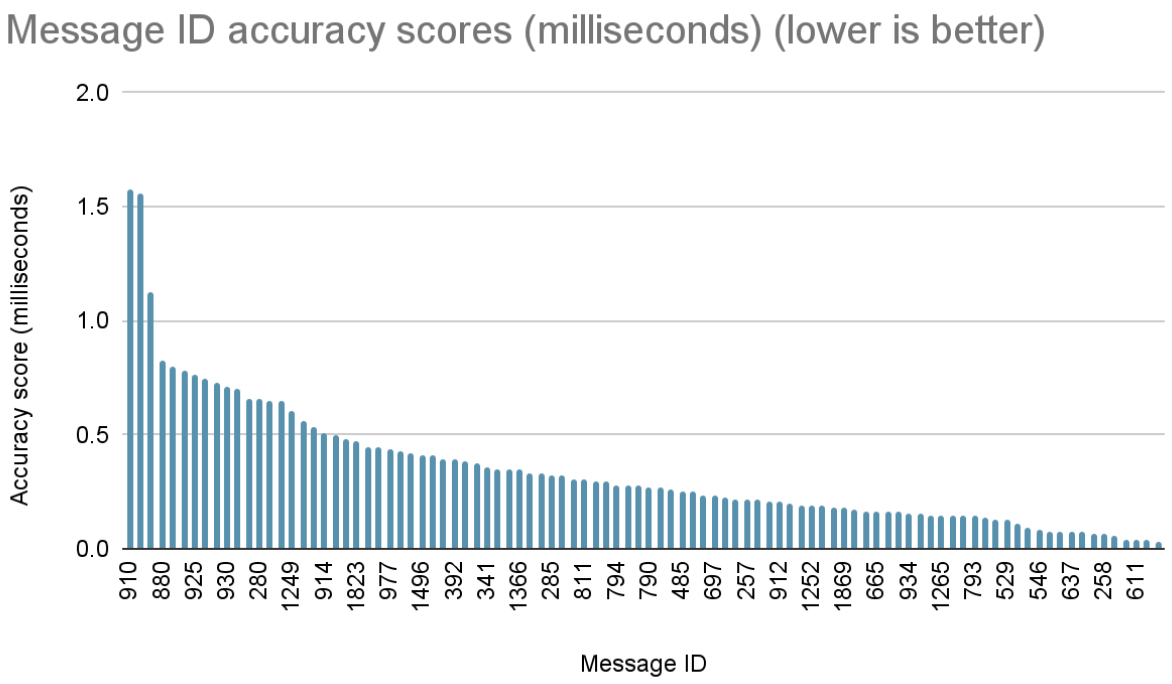
There are various methods for the client to use such as `all_message_ids()`, `period()`, and `timestamps()`, which return the data that their respective names would suggest they return. The `accuracy()` method takes a message_id (default value is the tree's message_id) and returns an "accuracy score". This is calculated by taking the average of the absolute-value differences between each gap between messages sent & the message's expected

period; therefore a lower accuracy score indicates better / more-on-time performance. `accuracy_report()` returns a dictionary mapping each message_id to its accuracy score.

Finally, I wrote 2 functions for client use: the first of which takes 2 file names and returns a `LogTree` object; the second of which takes the file name of a CSV file and a dictionary, and writes the dictionary data into the CSV. The former works as long as both CSVs are formatted similar to `logdata.csv` and `periods.csv`, with 2 columns, the first row as a header row (which the code ignores), and Message IDs in the appropriate column. The latter function is useful for extracting data obtained from the code I wrote.

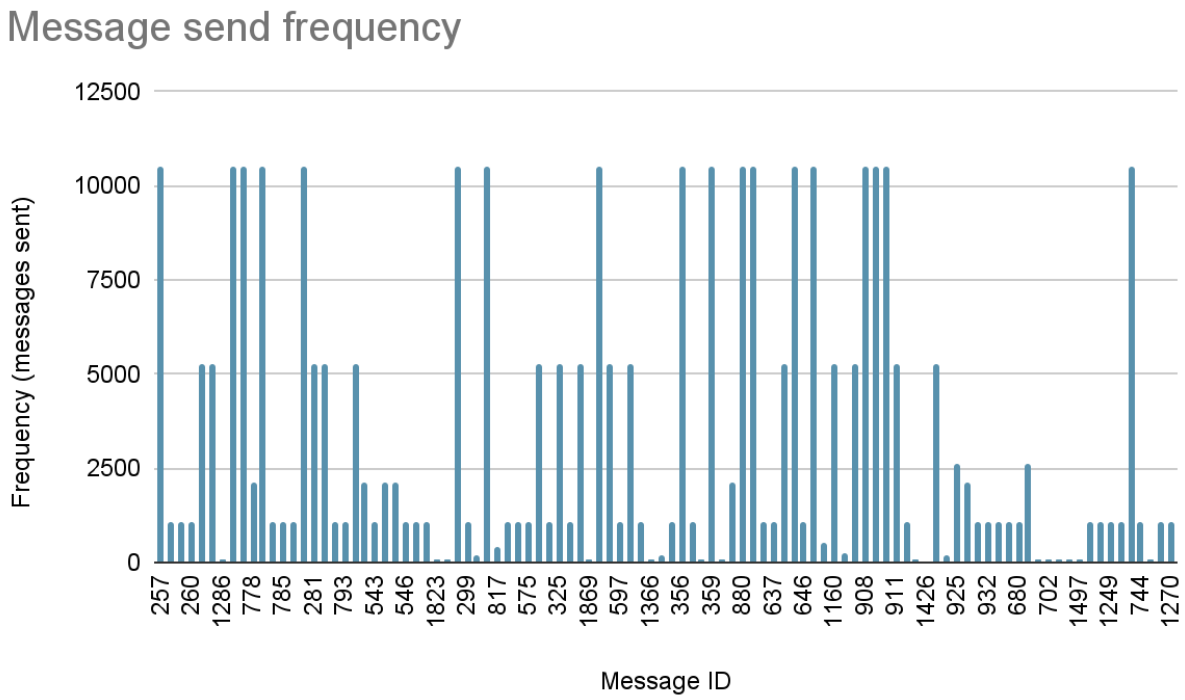
Part 2 of 2: Findings

Using the code I wrote, I was able to extract the following information about message jitter in the data provided to me. Below is a chart of each message ID's accuracy score in milliseconds:



It is to be noted that 4 message IDs were omitted for having anomalously high (inaccurate) accuracy scores, with scores of 6.36, 39.9, 750, and 1000 (rounded). The explanation for the last one, for example, is that the period for message ID 1426 is listed as 0 (which doesn't make sense); however if you look at its timestamps, it's clear that the period is meant to be 1000. Likewise with ID 817, whose period is seemingly mistakenly listed as 1000 when its timestamps reveal each gap is closer to 250. It is quite likely in my opinion that in addition to the 4 omitted messages, the 3 left-most messages on the above chart are also anomalous results, as they do not follow the overall trend you can see with the rest of the chart.

In addition to accuracy, I also extracted data on message frequency (how frequently each ID sent messages). As you can see below, it appears that almost all messages adopt 1 of several frequencies: 10,514, 5256, 2102, 1051, or 105. It's also worth noting that the frequency of messages sent often deviates from the norm by 1 message. For example, many message frequencies are actually 10,513 or 10,515 – something you can't tell from looking at the graph.



To conclude:

This is just an overview of my code and findings; I'd be more than happy to further discuss my ideas and thought process in an interview. Thank you for reading!