Coverage for **metric_generator.py** : 84%

163 statements | 137 run | 26 missing | 0 excluded

```
1
2  # Copyright 2020 Adam Campbell, Seth Hall, Andrew Ensor
3  # Copyright 2020 High Performance Computing Research Laboratory, Auckland University of Technology (AUT)
4
5  # Redistribution and use in source and binary forms, with or without
6  # modification, are permitted provided that the following conditions are met:
7
8  # 1. Redistributions of source code must retain the above copyright notice,
9  # this list of conditions and the following disclaimer.
10
11 # 2. Redistributions in binary form must reproduce the above copyright
12 # notice, this list of conditions and the following disclaimer in the
13 # documentation and/or other materials provided with the distribution.
14
15 # 3. Neither the name of the copyright holder nor the names of its
16 # contributors may be used to endorse or promote products derived from this
17 # software without specific prior written permission.
18
19 # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
20 # AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
21 # IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
22 # ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
23 # LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
24 # CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
25 # SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
26 # INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
27 # CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
28 # ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
29 # POSSIBILITY OF SUCH DAMAGE.
30
31 import numpy as np
32 import time
33 from influxdb import InfluxDBClient
34
35 class MetricGenerator:
36
37     # initialize Metric generator giving valid operands/operators for metrics
38     # also sets up an InfluxDBClient connection to locanl host
39     # ToDo make this a custom config connection
40     def __init__(self, num_channels, num_baselines, num_pols, to_database):
41         self.metric_list = []
42         self.valid_operands = ["real", "imaginary", "amplitude", "phase"]
43         self.valid_operators = ["min", "max", "mean", "variance"]
44         self.num_channels = num_channels
45         self.num_baselines = num_baselines
46         self.num_pols = num_pols
47         self.database = "gleam"
48         self.token = "need to generate this in future for HTTPS!"
49         self.client = InfluxDBClient(host='localhost', port=8086, database=self.database)
50         self.to_database = to_database
51
52     # Adds new instance of metric to track, assuming valid operator and operand.
53     # Note: Could benefit from validation of filters
54     def add_metric_instance(self, channel_filter, baseline_filter, pol_filter, operand,
55         operator, num_timesteps_to_process, start_timestamp = None, end_timestamp = None):
56
57         if operand not in self.valid_operands or operator not in self.valid_operators:
58             print("ERR: Invalid operand ({0:s}) or operator ({1:s}) supplied when adding new Metric".format(operand, operator))
59             return
60
61         self.metric_list.append(self.Metric(channel_filter, baseline_filter, pol_filter, operand, operator,
62             num_timesteps_to_process, start_timestamp, end_timestamp))
63
64     # Dispatches incoming payload to each metric being tracked for processing, the data gets saved to file or database.
65     # Maintains a list of metrics to be removed and dispatched after full metric list cycled
66     # Temp list is used to not mess with the for loop.
67     def process_payload(self, payload, start_channel, end_channel, timestep_timestamp):
68
69         to_be_removed = []
70
71         # Process payload across each metric
72         for m in self.metric_list:
73
74             # Has the metric started?
75             if m.start_timestamp <= time.time():
76
77                 if timestep_timestamp > m.timestep_timestamp:
78                     m.timestep_timestamp = timestep_timestamp
79                     m.timesteps_processed += 1
80
81                 if m.timesteps_processed > m.num_timesteps_to_process:
82                     m.timesteps_processed -=1
83                     if self.to_database:
84                         self.dispatch_metric_db(m)
85                     else:
86                         self.dispatch_metric_file(m)
87
88                     m.timestep_timestamp = timestep_timestamp
89                     m.timesteps_processed += 1
90
91
92                 #Process payload and determine if we have processed all anticipated payloads for current timestep
93                 m.process_payload(payload, start_channel, end_channel, timestep_timestamp)
```

```
94
95                  # Mark for destruction
96                  if m.end_timestamp <= time.time():
97                      to_be_removed.append(m)
98
99          # Destroy any expired metric instances
100         for m in to_be_removed:
101             self.metric_list.remove(m)
102
103     # Sends computed metric output to file.
104     # https://stackoverflow.com/a/3685339
105     # Saves three-dimensional matrix into file as a series of two-dimensional matrices
106     def dispatch_metric_file(self, metric):
107         print("Dispatching metric to file")
108         metric.post_processing()
109         file_name = "test_data/metric_output_" + metric.operand + "_" + metric.operator + ".txt"
110         with open(file_name, 'w') as outfile:
111             outfile.write('# Array shape: {0}\n'.format(metric.metric_data.shape))
112             for data_slice in metric.metric_data:
113                 np.savetxt(outfile, data_slice, fmt="%f")
114                 outfile.write('# New slice\n')
115
116         metric.reset_metric()
117
118     # Converts polarization into baseline string value
119     def getPol(self, polIndex, metric_output):
120         pol = metric_output.pol_filter[polIndex]
121         if pol == 0:
122             return "XX"
123         elif pol == 1:
124             return "XY"
125         elif pol == 2:
126             return "YX"
127         elif pol == 3:
128             return "YY"
129         else:
130             return "unknown"
131
132     # Sends the computed metric to influx DB database, the influxDB field is the chosen operator and influx value is the operand and channel number/start channel).
133     # Baseline and polarization and channel range are key tag values.
134     def dispatch_metric_db(self, metric):
135         print("Dispatching metric to influx DB")
136         metric.post_processing()
137         data = []
138         for chan in range(metric.metric_data.shape[0]):
139             for bl in range(metric.metric_data.shape[1]):
140                 for pol in range(metric.metric_data.shape[2]):
141                     data.append("{operator},chst={channel},chrg={range},bl={baseline},pol={polarization} {operand}={vis} {timestamp}".format(operator=metric.operator,
142                     channel='%0*d'%(6,metric.channel_filter[chan][0]),
143                     range=metric.channel_filter[chan][1] - metric.channel_filter[chan][0],
144                     baseline=metric.baseline_filter[bl],
145                     polarization=self.getPol(pol,metric),
146                     operand=metric.operand,
147                     vis=metric.metric_data[chan][bl][pol],
148                     timestamp=int(metric.timestep_timestamp*1000)))
149
150         self.client.write_points(data, database=self.database, time_precision='ms',batch_size=metric.metric_data.shape[0], protocol='line')
151
152         metric.reset_metric()
153
154     # https://stackoverflow.com/a/3685339
155     # Loads three-dimensional matrix from file from a series of two-dimensional matrices
156     def load_metrics_from_file(self, file_path, dimensions):
157         metric_data = np.loadtxt(file_path)
158         return metric_data.reshape(dimensions)
159
160     # Defines a singular metric instance for processing/tracking of
161     # desired metrics for incoming streams of visibility data (3D numpy array)
162     class Metric:
163
164         def __init__(self, channel_filter, baseline_filter, pol_filter, operand, operator,
165             num_timesteps_to_process, start_timestamp = None, end_timestamp = None):
166
167             self.channel_filter = channel_filter
168             self.baseline_filter = baseline_filter
169             self.pol_filter = pol_filter
170             self.num_timesteps_to_process = num_timesteps_to_process
171             self.timesteps_processed = 0
172             self.operand = operand.casefold()
173             self.operator = operator.casefold()
174             self.start_timestamp = start_timestamp
175             self.end_timestamp = end_timestamp
176             self.timestep_timestamp = 0
177
178             num_channels_intervals = self.channel_filter.shape[0]
179             num_baselines_to_process = self.baseline_filter.shape[0]
180             num_pols_to_process = self.pol_filter.shape[0]
181             self.dimensions = (num_channels_intervals, num_baselines_to_process, num_pols_to_process)
182             self.allocate_metric_buffer()
183
184             if start_timestamp is None: # user did not define during initialization
185                 self.start_timestamp = time.time()
186
187             if end_timestamp is None:
188                 self.end_timestamp = self.start_timestamp + (60 * 60) # defaults to 1 hour after
189
190         # Processes incoming payload(s) over time, computing requested operation
191         # and stores internally until the requested number of payloads have been
```

```
192         # processed. Performs compression across channels, and filters (ignores)
193         # baselines and pols not included in baseline and pol filters.
194         def process_payload(self, payload, start_channel, end_channel, timestep_timestamp):
195 
196             #print("start channel %d timestamp %d " %(start_channel, timestep_timestamp))
197             # For each channel range tuple in chan filter
198             chan_counter = 0
199             for chan_fil_interval in self.channel_filter:
200                 # For range in channel range tuple
201 
202                 for chan_interval in range(max(chan_fil_interval[0],start_channel), min(chan_fil_interval[1],end_channel)):
203                     baseline_counter = 0
204                     # For each baseline not ignored
205                     for baseline in self.baseline_filter:
206                         pol_counter = 0
207                         # For each pol not ignored
208                         for pol in self.pol_filter:
209                             payload_data = self.get_metric_operand(payload[chan_interval-start_channel][baseline][pol])
210                             previous_data = self.metric_data[chan_counter][baseline_counter][pol_counter]
211                             self.metric_data[chan_counter][baseline_counter][pol_counter] = self.perform_metric_op(previous_data, payload_data)
212                             pol_counter += 1
213 
214                         baseline_counter += 1
215 
216                 chan_counter += 1
217 
218         # Performs post processing of processed payloads before dispatching to file (or time-series database)
219         # Mean: calculate the overall average of the total processed payloads (dependant on filters)
220         # Variance: calculate the overall variance of the total processed payloads (dependant on filters)
221         # Min/Max: no post processing required
222         def post_processing(self):
223 
224             if self.operator == "mean":
225                 chan_counter = 0
226                 for chan_fil_interval in self.channel_filter:
227                     num_samples_per_interval = chan_fil_interval[1] - chan_fil_interval[0]
228                     x = (num_samples_per_interval * self.timesteps_processed)
229                     if x == 0:
230                         x = 1
231                     self.metric_data[chan_counter] /= x
232                     chan_counter += 1
233             elif self.operator == "variance":
234                 buffer = np.zeros(self.metric_data.shape, dtype=np.double)
235                 chan_counter = 0
236                 for chan_fil_interval in self.channel_filter:
237                     num_samples_per_interval = chan_fil_interval[1] - chan_fil_interval[0]
238                     val_sum_square = self.metric_data[chan_counter].imag / (num_samples_per_interval * self.timesteps_processed)
239                     val_sum = (self.metric_data[chan_counter].real / (num_samples_per_interval * self.timesteps_processed))**2
240                     buffer[chan_counter] = val_sum_square - val_sum
241                     chan_counter += 1
242                 self.metric_data = buffer
243             else: # ie; min or max
244                 pass
245 
246         def reset_metric(self):
247             self.allocate_metric_buffer()
248             self.timesteps_processed = 0
249             self.timestep_timestamp = 0
250 
251         # Allocates buffer with suitable datatype for specific metric being processed
252         def allocate_metric_buffer(self):
253 
254             if self.operator == "min":
255                 self.metric_data = np.full(self.dimensions, np.inf, np.double)
256             elif self.operator == "max":
257                  self.metric_data =  np.full(self.dimensions, np.NINF, dtype=np.double)
258             elif self.operator == "variance":
259                 self.metric_data =  np.zeros(self.dimensions, dtype=np.complex128)
260             else:
261                 self.metric_data =  np.zeros(self.dimensions, dtype=np.double)
262 
263         # Performs specific metric operation against processed
264         # payloads so far and the most recent payload received
265         def perform_metric_op(self, previous_data, payload_data):
266 
267             if self.operator == "min":
268                 return np.minimum(previous_data, payload_data)
269             elif self.operator == "max":
270                 return np.maximum(previous_data, payload_data)
271             elif self.operator == "mean":
272                 return previous_data + payload_data
273             elif self.operator == "variance":
274                 temp_real = previous_data.real + payload_data
275                 temp_imag = previous_data.imag + payload_data**2
276                 return temp_real + temp_imag * 1j
277 
278         # Gets or calculates the desired value requested by "this" metrics operand
279         def get_metric_operand(self, payload):
280 
281             if self.operand == "real":
282                 return payload.real
283             elif self.operand == "imaginary":
284                 return payload.imag
285             elif self.operand == "amplitude":
286                 return np.linalg.norm(payload) # magnitude
287             elif self.operand == "phase":
288                 return np.angle(payload) # arctan(im/re)
289 
```

```
290         # Basic To String for debugging purposes
291         def __str__(self):
292             return str(self.__class__) + ": " + np.array2string(self.metric_data)
```

« index coverage.py v5.2.1, created at 2020-08-14 09:55 +1200