Anomaly Detection in HTTP Logs

This sample notebook demonstrates working with HTTP request logs data stored in BigQuery.

Google Cloud Logging in the Google Cloud Platform makes it simple to export HTTP request logs from AppEngine applications directly into BigQuery for further analysis. This log data includes information such as the requested resource, HTTP status code, etc. One possible use of these logs is to mine them as they are collected to detect anomalies in response latency, since this can be a signal for some unexpected deployment issue.

The sample data used in this notebook is similar to AppEngine logs. It represents anonymized HTTP logs from a hypothetical application.

Related Links:

- Cloud Logging (https://cloud.google.com/logging/docs/)
- BigQuery (https://cloud.google.com/bigquery/what-is-bigquery)
- Pandas (http://pandas.pydata.org/) for data analysis
- Matplotlib (http://matplotlib.org/) for data visualization

```
from __future__ import division
import google.datalab.bigquery as bq
import matplotlib.pyplot as plot
import numpy as np
```

Understanding the Logs Data

It's helpful to inspect the dataset, the schema, and a sample of the data we're working with. Usually, logs are captured as multiple tables within a dataset, with new tables added per time window (such as daily logs).

```
%bq tables list --project cloud-datalab-samples --dataset httplogs
```

- BigQuery Table - name: cloud-datalab-samples.httplogs.logs_20140615
- BigQuery Table - name: cloud-datalab-samples.httplogs.logs_20140616
- BigQuery Table - name: cloud-datalab-samples.httplogs.logs_20140617
- BigQuery Table - name: cloud-datalab-samples.httplogs.logs_20140618
- BigQuery Table - name: cloud-datalab-samples.httplogs.logs_20140619
- BigQuery Table - name: cloud-datalab-samples.httplogs.logs_20140620

```
%bq tables describe -n cloud-datalab-samples.httplogs.logs_20140615
```

```
%%bq query -n logs
SELECT timestamp, latency, status, method, endpoint
FROM `cloud-datalab-samples.httplogs.logs_20140615`
ORDER by timestamp
```

```
%%bq sample --query logs --count 7
```

timestamp	latency	status	method	endpoint
2014-06-15 07:00:00.003772	122	200	GET	Interact3
2014-06-15 07:00:00.428897	144	200	GET	Interact3
2014-06-15 07:00:00.536486	48	200	GET	Interact3
2014-06-15 07:00:00.652760	28	405	GET	Interact2
2014-06-15 07:00:00.670100	103	200	GET	Interact3
2014-06-15 07:00:00.834251	121	405	GET	Interact2
2014-06-15 07:00:00.943075	28	200	GET	Other

(rows: 7, time: 0.2s, cached, job: job_FZlEupE0IV2DlD_nuS1pYJx1lsk)

Transforming Logs into a Time Series

We're going to build a timeseries over latency. In order to make it a useful metric, we'll look at 99th percentile latency of requests within a fixed 5min window using this SQL query issued to BigQuery (notice the grouping over a truncated timestamp, and quantile aggregation).

```
%%bq query -n timeseries
SELECT DIV(UNIX_SECONDS(timestamp), 300) * 300 AS five_minute_window,
    APPROX_QUANTILES(latency, 99)[SAFE_ORDINAL(99)] as latency
FROM `cloud-datalab-samples.httplogs.logs_20140615`
WHERE endpoint = 'Recent'
GROUP BY five_minute_window
ORDER by five_minute_window
```

```
%%bq sample --query timeseries --count 10
```

five_minute_window	latency
1402815600	427
1402815900	329
1402816200	293
1402817400	242
1402818000	332
1402818300	288
1402818900	299
1402819800	294
1402820400	111
1402821000	361

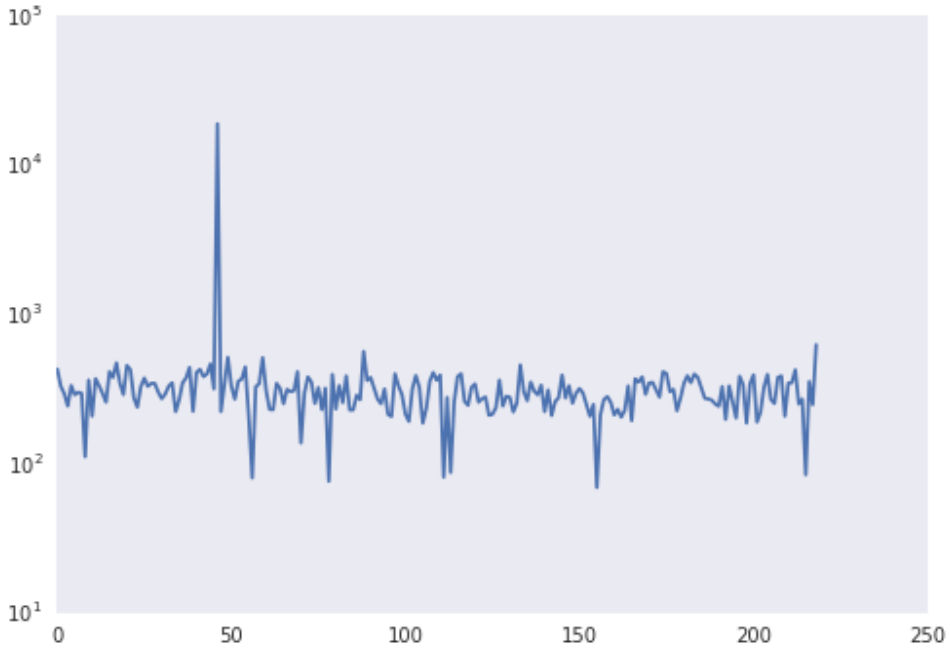
(rows: 10, time: 0.2s, cached, job: job_eEP6Llqt0lpZVTpiOVGMLJ4P9A4)

Visualizing the Time Series Data

Its helpful to visualize the data. In order to visualize this timeseries, we'll use python, pandas and matplotlib.

```
# Execute and convert the results to a Pandas dataframe
timeseries_df = timeseries.execute(output_options=bq.QueryOutput.dataframe()).result()

timeseries_values = timeseries_df['latency'].values
timeseries_len = len(timeseries_values)
```

```
plot.plot(np.array(range(timeseries_len)), timeseries_values)
plot.yscale('log')
plot.grid()
```



Anomaly Detection

A visual inspection of the chart, above, highlights the obvious anomalies, but we want to construct something that can inspect the data, learn from it, and detect anomalies in an automated way.

Anomaly Detector

The code below is a simple anomaly detector created for purposes of a sample. It uses a simple algorithm that tracks mean and standard deviation. You can give it a value indicating the number of instances is uses to train a model before it starts detecting anomalies. As new values are passed to it, it continues to update the model (to account for slowly evolving changes) and report anomalies. Any value that is off from the mean by 3x the standard deviation is considered as an anomaly.

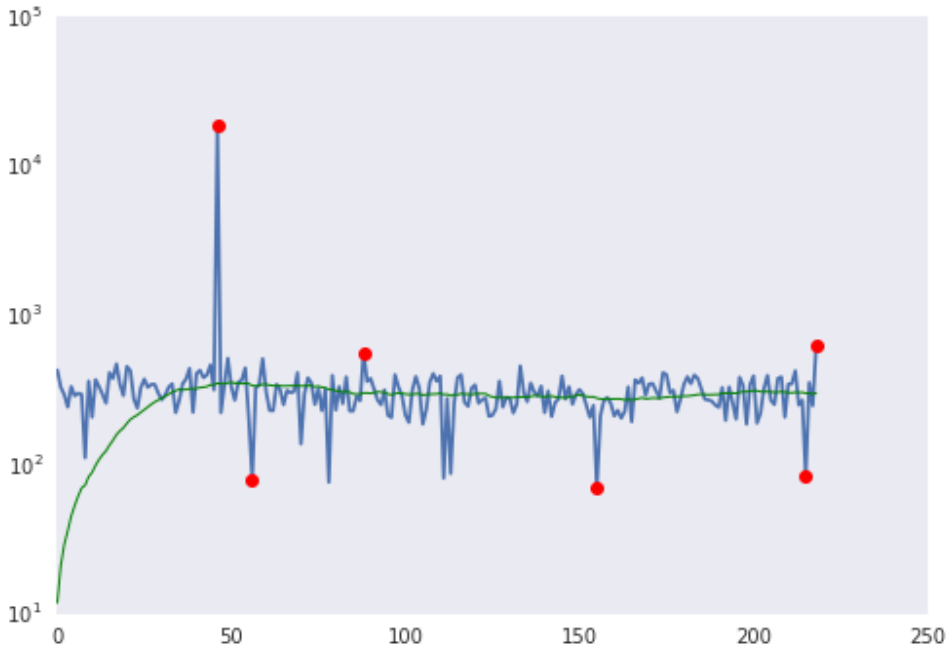
```
class AnomalyDetector(object):

  def __init__(self, window = 10):
    self._index = 0
    self._window = window
    self._values = np.zeros(window)
    self._valuesSq = np.zeros(window)
    self._mean = 0
    self._variance = 0
    self._count = 0

  def observation(self, value):
    anomaly = False

    threshold = 3 * np.sqrt(self._variance)
    if self._count > self._window:
      if value > self._mean + threshold:
        value = self._mean + threshold
        anomaly = True
      elif value < self._mean - threshold:
        value = self._mean - threshold
        anomaly = True
    else:
      self._count += 1

    prev_value = self._values[self._index]
    self._values[self._index] = value
    self._valuesSq[self._index] = value ** 2
    self._index = (self._index + 1) % self._window

    self._mean = self._mean - prev_value / self._window + value / self._window
    self._variance = sum(self._valuesSq) / self._window - (self._mean ** 2)

    return anomaly, self._mean
```

With the anomaly detector implemented, let's run the timeseries through it to collect any anomalies and the expected mean along each point.

```
anomalies = np.zeros(timeseries_len)
means = np.zeros(timeseries_len)

anomaly_detector = AnomalyDetector(36)

for i, value in enumerate(timeseries_values):
  anomaly, mean = anomaly_detector.observation(value)

  anomalies[i] = anomaly
  means[i] = mean
```

Then, plot the same time series, but overlay the anomalies and the mean values as well.

```
ticks = np.array(range(timeseries_len))

plot.plot(ticks, timeseries_values)
plot.plot(ticks[anomalies == 1], timeseries_values[anomalies == 1], 'ro')
plot.plot(ticks, means, 'g', linewidth = 1)
plot.yscale('log')
plot.grid()
```



Going Beyond the Sample

This sample demonstrated how to use SQL to transform raw log data into an interesting timeseries, then apply Python logic to detect and visualize anomalies.

Here are some next steps that to make this exercise more useful.

- Build an initial model based on training data over a representative time, then store that model.
- Parameterize the source table of the logs. You'll want to use a table that is date based, and use a table decorater to convert the last N minutes of log data into a single timeseries point.
- Pass the resulting timeseries point into an anomaly detector that uses the stored model, then publishes anomalies to a pub/sub topic.

The notebook allows you to work with your data to explore and understand it, working through the steps to derive data and arrive at a proof of concept that you can take to build a final solution.