Incremental Machine Learning by Example: Detecting Suspicious Activity with Zeek Data Streams, River, and JA3 Hashes

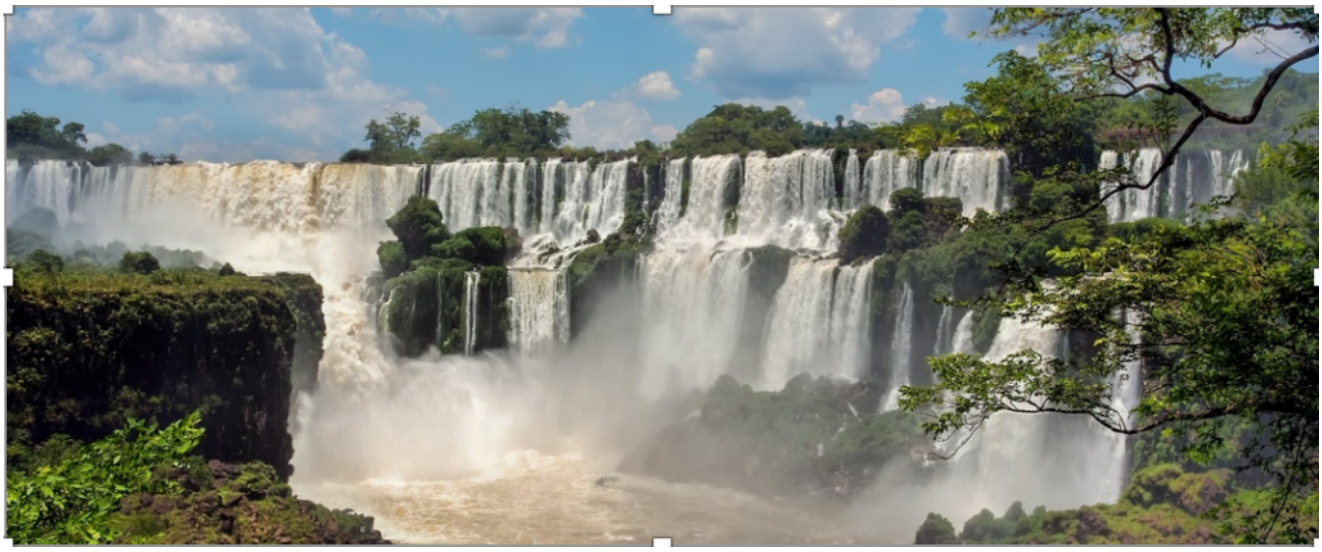
research.nccgroup.com/2021/06/14/incremental-machine-leaning-by-example-detecting-suspicious-activity-with-zeek-data-streams-river-and-ja3-hashes/

June 14, 2021

tl:dr

Incremental Learning is an extremely useful machine learning paradigm for deriving insight into cyber security datasets. This post provides a simple example involving JA3 hashes showing how some of the foundational algorithms that enable incremental learning techniques can be applied to novelty detection (the first time something has happened) and outlier detection (rare events) on data streams derived from Zeek[i]. Creation of these features is extremely useful for further modelling and for facilitating effective anomaly and behavioural alerting.

Working with Unbounded Streams



The beauty and terror of having lots of streams

In academic papers the machine learning models that are used to address challenges in cyber security are overwhelmingly trained on bounded datasets – where the dataset has a clear start and end. However, many of the data sets that data scientists work with when helping cyber security specialists trying to defend a network, investigate an incident or attack a customer are unbounded data streams – we have a start to the data stream, but we are unaware of when it ends.

This creates a challenge.

In the context of data streams, it is assumed that data can change over time[ii]. Unforeseen changes to the properties of the data makes statically trained models at best less accurate and at worst irrelevant. Even in the most agile of engineering environments there is a cost, in time and effort, of taking a model offline, retraining the model and then placing the model back into production.

Online or incremental machine learning is a way of overcoming this challenge. Given the advantages of the insights and the speed at which they can be calculated it is surprising that incremental learning does not garner greater attention when addressing challenges in cyber security.

What is Incremental Learning?

Incremental learning is a method in machine learning in which data becomes available in a sequential order[iii]. The technique is often defined by how it differs from batch learning. Batch learning generates the best predictor by learning on the entire training data set at once. In incremental learning predications are updated at each step.

Using incremental learning techniques, machine learning models can adapt to constantly arriving data streams. A subset of algorithms facilitates this process. Welford's method, proved in 1962 in Macclesfield, is a good example of one such algorithm[iv].

Welford's method was innovative as it proved that variance can be calculated in fewer steps than would normally be required to calculate standard deviation. This simple feat is achieved by the use of recursion – as new data arrives a variable is updated by reference to itself.

Consider the following problem of maintaining a running mean.

> You have an array: [2, 3, 5, 7, 11, 13, 17, 19]
>
> To create a mean, you sum all the values in the array and them divide by the number of elements: [2, 3, 5, 7, 11, 13, 17, 19] / 8
>
> If a new element was added to the array to calculate a new mean you could repeat the step above, sum all the values and divide by the number of elements: [2, 3, 5, 7, 11, 13, 17, 19, 23] / 9
>
> However, if you have a rapidly increasing data set a point will be reached when using this method to calculate the mean becomes prohibitively expensive. Keeping a potentially unbounded array in memory will eventually result in a lack of memory.
>
> An incremental learning approach to the problem of calculating a running mean is to only hold 2 values in memory:
> - The sum of all the elements in the array
> - A running count on the number of elements in the array
>
> When the next element arrives, we can observe recursion in action:
> - total_sum = total_sum + new_element
> - running_count = running_count + 1
> - current_mean = total_sum / running_count

Why is Incremental Learning Useful?

Some key benefits of an incremental learning approach include:

> The ability to infer statistics and obtain accurate machine learning predictions in near real time on affordable infrastructure, on unbounded datasets.
>
> The ability to be more selective with the data that is stored.
>
> Increased efficiency, which reduces requirements for complicated architectures, such as distributed compute, which minimises the attack surface.
>
> Bypassing problems often faced when attempting to use batch trained machine learning models in production settings. Broadly speaking, incremental learning models can be taught to adapt; batch learning models have to be retrained.

River: A Python Package for Incremental Learning

River[v], a Python package, is a recommended place to start if you want to use incremental machine learning in a project. Especially if you are unfamiliar, or have forgotten mathematical notation, looking at these algorithms in Python is a good way to grasp how they work and what they do. Max Halford's excellent overview of an early version of the package to PyData, Amsterdam is recommended viewing[vi].

Zeek + River: Doing More with JA3 Hashes

Zeek (formerly known as bro) is an open source and widely used tool for capturing, processing, and generating logs from network traffic; Zeek is great for generating unbounded datasets[vii]. Zeek comes with its own messaging library which can be accessed via a Python API[viii] [ix] [x]. Use of this API enables the creation of Python generators which can be updated in near real time with network derived data – such as JA3 hashes.

JA3 hashes, developed by the security team at Salesforce[xi], are a way of fingerprinting TLS applications (both clients and servers). A use of this fingerprint is to share a JA3 hash, identified as malicious, as an 'Indicator of Compromise' to further discover uses of undesirable software.

However, JA3 hashes of legitimate and malicious software sometimes overlap. Our Security Operations Centre Analyst team and our Incident Responders became quickly frustrated that a JA3 hash indicative of a malicious application on one network would frequently be the hash of a legitimate application on a separate, or even the same, network.

Feature Extraction: Classifying and Scoring JA3 Rarity

To utilise JA3 hashes instead of relying on string matching, additional features can be extracted from the data and incorporated into incremental learning models to help to identify suspicious activity in near real-time.

An example of such a feature is classifying whether the hash is rare for the network. Furthermore, if classified as rare, a score between 0 (most common in the rarity class) and 100 (most rare in the rarity class) is calculated. Some code, in its exploratory format, is

shared below to demonstrate this approach.

Rather than setting up an entire zeek pipeline synthetic data is created for this demonstration. The distribution of the data reflects the frequency of JA3 hashes from a commercial environment of approximately 2,000 devices over a 3-month time-period. In total 71,057,480 JA3 hashes were observed, of which 2,907 were unique JA3 hashes.

In the following list the first item in each tuple relates to the frequency – the number of times a JA3 hash was observed. The second item in the tuple relates to the number of unique JA3 hashes which had that corresponding frequency score. The JA3 hash distribution list is combined with a random JA3 hash to produce synthetic data.

```
import random
import math
import secrets
import pandas as pd
import numpy as np
from river import stats
from river import proba
from river import utils
```

```
JA3_hash_distribution = [(1, 557), (2, 264), (3, 200), (4, 195), (5, 56), (6, 55),
(7, 119), (8, 127), (9, 145), (10, 120), (11, 57), (12, 41), (13, 15), (14, 24), (15,
16), (16, 22), (17, 11), (18, 19), (19, 14), (20, 13), (21, 13), (22, 16), (23, 10),
(24, 7), (25, 11), (26, 8), (27, 8), (28, 7), (29, 7), (30, 5), (31, 2), (32, 7),
(33, 6), (34, 2), (35, 8), (36, 4), (37, 4), (38, 6), (39, 5), (40, 5), (42, 2), (43,
4), (44, 3), (45, 6), (46, 3), (47, 3), (48, 1), (49, 1), (50, 4), (51, 3), (52, 3),
(53, 2), (54, 2), (55, 2), (56, 5), (57, 1), (58, 4), (59, 5), (60, 3), (61, 4), (62,
2), (63, 2), (64, 3), (65, 3), (66, 3), (67, 2), (68, 2), (69, 2), (70, 1), (71, 2),
(72, 2), (73, 3), (74, 2), (75, 2), (76, 2), (77, 4), (78, 3), (79, 1), (80, 2), (81,
1), (82, 2), (83, 3), (84, 1), (85, 2), (87, 2), (88, 2), (89, 1), (91, 1), (93, 1),
(94, 5), (95, 5), (97, 2), (98, 3), (101, 2), (103, 1), (104, 1), (106, 2), (108, 1),
(109, 1), (111, 2), (112, 2), (113, 3), (114, 3), (115, 4), (116, 1), (117, 1), (118,
1), (119, 4), (120, 3), (121, 1), (122, 1), (123, 2), (124, 3), (125, 3), (126, 1),
(127, 2), (128, 5), (130, 4), (131, 2), (132, 1), (134, 1), (135, 1), (136, 3), (138,
1), (139, 2), (140, 2), (143, 2), (144, 4), (146, 1), (148, 3), (150, 1), (151, 1),
(152, 2), (153, 1), (154, 1), (155, 1), (156, 3), (157, 2), (159, 1), (160, 1), (161,
1), (162, 2), (163, 1), (165, 1), (166, 1), (168, 4), (169, 1), (172, 1), (176, 1),
(177, 1), (179, 2), (180, 2), (181, 1), (182, 2), (188, 2), (189, 1), (190, 1), (192,
1), (199, 1), (202, 2), (203, 2), (212, 2), (214, 1), (215, 1), (216, 1), (221, 1),
(223, 1), (228, 1), (232, 1), (233, 1), (234, 1), (238, 2), (240, 1), (242, 1), (243,
1), (249, 1), (253, 1), (254, 1), (260, 1), (265, 1), (268, 1), (270, 1), (271, 1),
(272, 1), (278, 1), (279, 1), (280, 1), (281, 1), (282, 1), (288, 1), (289, 1), (291,
1), (295, 1), (296, 1), (300, 1), (301, 1), (302, 1), (303, 1), (307, 1), (309, 1),
(311, 1), (315, 1), (330, 1), (331, 1), (333, 1), (339, 1), (352, 1), (353, 1), (358,
1), (368, 1), (372, 2), (380, 1), (382, 1), (383, 1), (387, 1), (388, 1), (394, 1),
(395, 1), (399, 1), (400, 2), (406, 2), (407, 2), (412, 1), (414, 1), (417, 1), (421,
1), (422, 2), (425, 1), (426, 1), (427, 1), (429, 1), (431, 1), (436, 1), (439, 2),
(442, 1), (449, 1), (459, 1), (463, 1), (473, 1), (475, 1), (476, 1), (477, 1), (480,
1), (482, 1), (487, 1), (489, 2), (490, 1), (492, 1), (493, 1), (502, 1), (503, 1),
(504, 1), (508, 1), (512, 1), (514, 1), (517, 1), (518, 1), (534, 1), (539, 1), (544,
1), (545, 1), (557, 1), (566, 1), (574, 1), (576, 1), (580, 1), (584, 2), (607, 1),
(608, 1), (617, 2), (624, 2), (639, 1), (645, 1), (647, 1), (661, 1), (676, 1), (677,
1), (679, 1), (688, 1), (692, 1), (696, 1), (703, 1), (709, 1), (711, 1), (726, 1),
(733, 1), (754, 1), (755, 1), (757, 1), (758, 1), (768, 1), (776, 1), (781, 1), (785,
1), (799, 1), (805, 1), (820, 1), (822, 1), (829, 1), (830, 1), (846, 1), (878, 1),
(882, 1), (883, 1), (888, 2), (889, 1), (890, 1), (896, 1), (906, 1), (912, 1), (917,
1), (941, 1), (965, 1), (981, 1), (1004, 1), (1010, 2), (1029, 1), (1042, 1), (1048,
1), (1056, 1), (1081, 1), (1101, 1), (1121, 2), (1126, 1), (1156, 1), (1179, 1),
(1183, 1), (1210, 1), (1220, 1), (1222, 1), (1289, 1), (1293, 1), (1294, 1), (1297,
1), (1298, 1), (1299, 1), (1315, 1), (1325, 1), (1425, 1), (1437, 1), (1529, 1),
(1530, 1), (1601, 1), (1606, 1), (1647, 1), (1696, 1), (1702, 1), (1705, 1), (1732,
1), (1836, 1), (1843, 1), (1860, 1), (1862, 1), (1886, 1), (1889, 1), (1918, 1),
(2034, 1), (2036, 1), (2158, 1), (2174, 1), (2220, 1), (2248, 1), (2289, 1), (2338,
1), (2362, 1), (2428, 1), (2432, 1), (2434, 1), (2439, 1), (2451, 1), (2457, 1),
(2490, 1), (2494, 1), (2628, 1), (2631, 1), (2636, 1), (2660, 1), (2662, 1), (2843,
1), (2892, 1), (2915, 1), (2931, 1), (2951, 1), (2962, 1), (2990, 1), (3004, 1),
(3045, 1), (3180, 1), (3187, 1), (3204, 1), (3206, 1), (3240, 1), (3254, 1), (3285,
1), (3416, 1), (3424, 1), (3542, 1), (3571, 1), (3646, 1), (3679, 1), (3715, 1),
(3839, 1), (3862, 1), (3892, 1), (3913, 1), (3934, 1), (3978, 1), (4024, 1), (4050,
1), (4095, 1), (4217, 1), (4344, 1), (4374, 1), (4518, 1), (4762, 1), (4846, 1),
(4905, 1), (5057, 1), (5073, 1), (5207, 1), (5348, 1), (5480, 1), (5484, 1), (5545,
1), (5629, 1), (5725, 1), (5802, 1), (5838, 1), (6171, 1), (6292, 1), (6309, 1),
(6366, 1), (6475, 1), (6550, 1), (6567, 1), (6651, 1), (6718, 1), (6754, 1), (6912,
1), (6913, 1), (6931, 1), (7043, 1), (7049, 1), (7180, 1), (7216, 1), (7246, 1),
(7309, 1), (7560, 1), (7786, 1), (7901, 1), (7928, 1), (8136, 1), (8169, 1), (8174,
```

```
1), (8936, 1), (9193, 1), (9204, 1), (9265, 1), (9455, 1), (9516, 1), (9882, 1),
(10269, 1), (10313, 1), (10390, 1), (10866, 1), (11087, 1), (11221, 1), (11307, 1),
(11647, 1), (11648, 1), (11667, 1), (11698, 1), (11714, 1), (11907, 1), (12817, 0),
(12913, 1), (13291, 1), (13414, 1), (14385, 1), (14913, 1), (16006, 1), (16353, 1),
(16372, 1), (16852, 1), (17511, 1), (17897, 1), (18213, 1), (18991, 1), (19559, 1),
(20104, 1), (21680, 1), (22139, 1), (22280, 1), (22568, 1), (22571, 1), (22702, 1),
(22956, 1), (23042, 1), (23223, 1), (24382, 1), (25283, 1), (25573, 1), (27819, 1),
(28241, 1), (30189, 1), (30607, 1), (33555, 1), (34208, 1), (34629, 1), (36851, 1),
(37422, 1), (38866, 1), (39448, 1), (40312, 1), (41026, 1), (41233, 1), (42905, 1),
(43977, 1), (49260, 1), (49743, 1), (50291, 1), (51877, 1), (54034, 1), (56779, 1),
(58525, 1), (58906, 1), (60612, 1), (67749, 1), (75869, 1), (80919, 1), (81567, 1),
(93041, 1), (94499, 1), (110338, 1), (118224, 1), (127034, 1), (135345, 1), (136999,
1), (151899, 1), (163236, 1), (168476, 1), (175232, 1), (178976, 1), (196220, 1),
(197064, 1), (217345, 1), (225238, 1), (227414, 1), (255540, 1), (255685, 1),(260018,
1), (266592, 1), (279776, 1), (283654, 1), (290762, 1), (301814, 1), (321114, 1),
(389111, 1), (401335, 1), (417942, 1), (437972, 1), (440219, 1), (448069, 1),
(507097, 1), (511055, 1), (581968, 1), (587674, 1), (615067, 1), (652445, 1),
(713196, 1), (874180, 1), (980906, 1), (1137667, 1), (1219190, 1), (1260749, 1),
(1276803, 1), (1546597, 1), (1682322, 1), (1986321, 1), (2038725, 1), (2772989, 1),
(2861818, 1), (4195102, 1), (8502133, 1), (24049793, 1)]

def create_JA3_hash(n_count, mock_distribution):
    """Generate a fictitious JA3 hash"""
    imaginary_JA3_hash = secrets.token_hex(nbytes=16)
    imaginary_JA3_hash = n_count * mock_distribution * (imaginary_JA3_hash,)
    return imaginary_JA3_hash

# loop through the hash_distribution and assign a fictitous JA3 hash
_data = []

for n_count, mock_distribution in JA3_hash_distribution:
    imaginary_JA3_hash = create_JA3_hash(n_count, mock_distribution)
    _data.append(imaginary_JA3_hash)

# flatten list of tuples into a list
synth_data = [flat_list for tuple_values in _data for flat_list in tuple_values]

# shuffle the data
ja3_stream = random.sample(synth_data, 71057480)
```

The following APIs from River are used to generate a meaningful rarity score on every JA3 hash:

- A streaming multinomial probability[xii]
- A streaming quantile[xiii]
- A streaming histogram[xiv]

```
#  Init incremental Multinomial distribution
p = proba.Multinomial()

# Init incremental Multinomial distribution for values in the 50th percentile
p2 = proba.Multinomial()

# Init incremetal identification of the 99.5th quantile in p
p_quantile_99_5 = stats.Quantile(0.995)

# Init histogram to calculate CDF of the PDF
hist = utils.Histogram()
```

Three functions are used to calculate the probability mass function (PMF) of the JA3 hash; identify whether the hash should be considered a rare hash or not; and then to create a running cumulative distribution functions (CDF) on the JA3 hashes in the rarity class. The CDF score is equated to the finalised rarity score associated with the JA3 hash. A purpose of this secondary rarity score is to overcome the “Second strong law of small numbers”[xv] and make the output humanly comprehensible.

```
def pmf_rarity_score(ja3, p):
    """Calculate a rarity score"""
    # update the probability mass function (pmf)
    p = p.update(ja3)
    # calculating rarity from the pmf, inversed and turned into a percentage
    p_rarity_score = (1 - p.pmf(ja3)) * 100
    return p_rarity_score


def rarity_classification(p_rarity_score, p_quantile_99_5):
    """Classify the JA3 as rare / not rare"""
    # update the running quantile
    p_quantile_99_5.update(p_rarity_score)
    rarity_class = np.absolute(p_quantile_99_5.get())
    # if rarity score is greater than the 99.5th percentile than the hash is rare
    if p_rarity_score > rarity_class:
        rare = True
    else:
        rare = False
    return rare


def cdf_of_rare_events(ja3, p2, hist):
    """Create a new feature by scoring the rare event"""
    # update the pmf of events in the rarity class
    p2 = p2.update(ja3)
    p2_rarity_score = (1 - p2.pmf(ja3)) * 100
    hist = hist.update(p2_rarity_score)
    for z, item in zip([p2_rarity_score], hist.iter_cdf([p2_rarity_score])):
        cdf = math.ceil(item * 100)
        alerting_features = ja3, p2_rarity_score, cdf
        return alerting_features
```

A for loop is used to replay the synthetic data as a stream. Note, this is a significant volume of data. Depending on your hardware it may take approximately 15 minutes. In practice, the cost of this calculation would be spread over three months. If impatient for the cell execution to finish, then interrupt the kernel, you will still have meaningful output.

```
ja3_rarity_feature = []

for ja3 in ja3_stream:
    p_rarity_score = pmf_rarity_score(ja3, p)
    rare = rarity_classification(p_rarity_score, p_quantile_99_5)
    if rare:
        alerting_features = cdf_of_rare_events(ja3, p2, hist)
        ja3_rarity_feature.append(alerting_features)
```

The next cell provides a starting point for beginning to explore the values of the feature extraction.

```
df2 = pd.DataFrame(ja3_rarity_feature)
df2[0].value_counts()

#  view how the rarity score of a JA3 changed over time
ja3_of_interest = "Replace with JA3 of interest"
df2[0].str.count(ja3_of_interest).sum()
df3 = df2.loc[df2[0] == ja3_of_interest]
df3 = df3.rename(columns={0:'JA3', 1:'initial_rarity_score', 2:
'finalized_rarity_score'})
df3
```

So What? Moving from Weird to Bad

A single feature can be interesting, especially when permanently stored, to aid with incident response, or to facilitate an initial SOC investigation. Nonetheless, a rarity score is a measure of weirdness, or novelty, not badness.

To move from weird to bad and ensure that these calculations have impact on improved detection further work is required. Some of the steps that are currently used to achieve this are outlined below.

> **Combine rarity and novelty scoring with rule detection.** A hash becomes more interesting if you have a known bad JA3 and it has been classified as new or rare.

Combine rarity with other features and fuzzy logic to create heuristics. For example, when attempting to identify command and control beaconing, other features of interest which are extracted include Beaconing Score, ASN rarity and External Domain rarity. In “Hunting for Beacons” Ruud van Luijk describes statistical techniques for calculating beacon scores[xvi]. Such scoring can also be calculated efficiently and accurately with the use of online algorithms. Hard coded limits on these features can be set, or the features can be combined with fuzzy logic techniques to create heuristics that alert on connections or achieved conditions that are observed within a specified time.

Further modelling. Rather than relying on pre-determined parameters other incremental learning methods can also be utilised to obtain an anomaly score from multiple features (multi*variate* data). One example of this method would be the use of an online version of the Mahalanobis Distance[xvii].

Automated investigation. In conjunction with our SOAR platform and in house Cyber Threat Intelligence techniques[xviii] alerting becomes a data point within a larger architecture whereby incremental learning techniques – often with a more supervised leaning, or deep learning techniques, can be used to classify events as being likely associated to malicious, rather than just suspicious, activity.

Conclusion

As a Data Science team at NCC Group and Fox-IT we have found that incremental learning adds significant value when handling unbounded data sets. The alerting output of these models is fast, reliable, and challenging-to-evade detection logic. Ongoing work continues developing and implementing incremental learning and other machine learning techniques to facilitate production worthy features and models which create detection alerting output with improved precision and recall metrics.

The Code

https://github.com/nccgroup/JA3_outlier

[i] https://zeek.org/

[ii] https://riverml.xyz/dev/examples/concept-drift-detection/

[iii] https://www.wikiwand.com/en/Online_machine_learning

[iv]https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=8153FDB64C81A77FAE4F8B3F675589CD?doi=10.1.1.302.7503&rep=rep1&type=pdf

[v] https://riverml.xyz/latest/

[vi] https://www.youtube.com/watch?v=P3M6dt7bY9U

[vii] Eric Ooi's blog series is very useful if you are new to setting up zeek https://www.ericooi.com/zeekurity-zen-zeries/

[viii] https://github.com/zeek/broker

[ix] This webinar from Dominic Charousset gives a helpful overview of how to use broker and upcoming changes in broker https://event.webinarjam.com/replay/24/yvwmyaqlcl9i96iw2

[x] Rather than using zeek broker Ben Bornholm's article on steaming zeek logs with KSQL provides an alternative approach https://holdmybeersecurity.com/2020/06/08/poc-using-ksql-to-enrich-zeek-logs-with-osquery-and-sysmon-data/

[xi] https://github.com/salesforce/ja3

[xii] https://riverml.xyz/dev/api/proba/Multinomial/

[xiii] https://riverml.xyz/dev/api/stats/Quantile/

[xiv] https://riverml.xyz/dev/api/utils/Histogram/

[xv] https://en.wikipedia.org/wiki/Strong_Law_of_Small_Numbers

[xvi] https://blog.fox-it.com/2020/01/15/hunting-for-beacons/

[xvii] https://scikit-learn.org/stable/auto_examples/covariance/plot_mahalanobis_distances.html

[xviii] https://blog.fox-it.com/2019/02/26/identifying-cobalt-strike-team-servers-in-the-wild/