AITK and DSDL Operationalization and Use Case Workshop

Lab Guide

This lab guide contains the hands-on exercises for the AITK and DSDL Operationalization and Use Case workshop. Before proceeding with these exercises, please ensure that you have a copy of the workshop slide deck, which will help to put into context the tasks you are carrying out.

Download the workshop slide deck: https://splk.it/MLTK-DSDL-Attendee

Prerequisites

In order to complete these exercises, you will need your own Splunk instance. Splunk's hands-on workshops are delivered via the Splunk Show portal and you will need a splunk.com account in order to access this.

If you don't already have a Splunk.com account, please create one here before proceeding with the rest of the workshop.

⚠️ Troubleshooting Connectivity

If you experience connectivity issues with accessing either your workshop environment or the event page, please try the following troubleshooting steps. If you still experience issues please reach out to the team running your workshop.

- **Use Google Chrome** (if you're not already)
- If the event page (i.e. *https://show.splunk.com/event/<eventID>*) didn't load when you clicked on the link, try **refreshing the page**
- **Disconnect from VPN** (if you're using one)
- **Clear your browser cache and restart your browser** (if using Google Chrome, go to: Settings > Privacy and security > Clear browsing data)
- **Try using private browsing mode** (e.g. Incognito in Google Chrome) to rule out any cache issues
- **Try using another computer** such as your personal computer - all you need is a web browser! Cloud platforms like AWS can often be blocked on corporate laptops.

Table of Contents

Overview

Welcome to the AITK and DSDL Operationalization and Use Case Workshop lab guide! This workshop consists of 5 modules. The modules are designed to be independent of each other, however for the full experience we recommend going through each module in order.

A few quick notes about how the exercise guide is formatted:

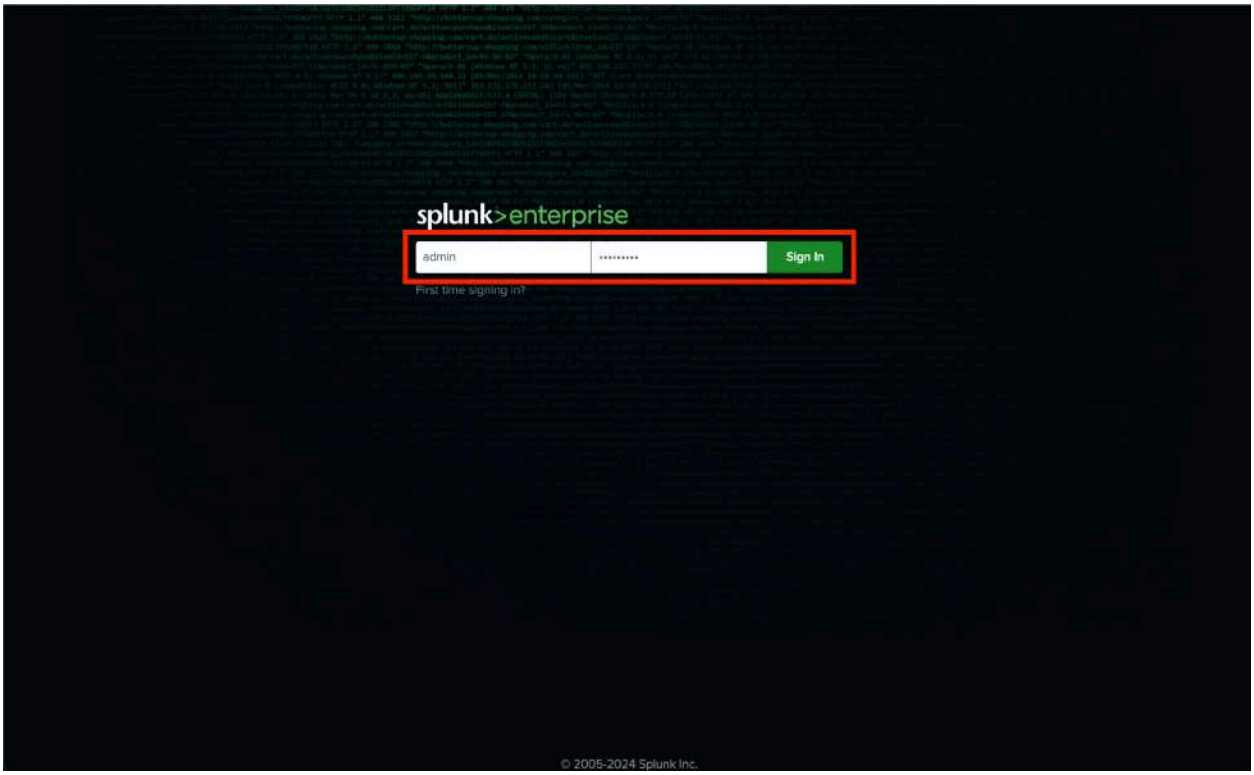
- Mono-spaced text `like this` and code blocks indicate specific callouts you need to type or pay attention to.
- Bolded text **like this** indicates an action you need to take such as clicking a button in a browser window.
- The screenshots in the lab use the light theme so it's easier to read in the instructions. However, your environment might be configured to use the dark theme. If you would like to change your interface theme setting, you're welcome to do so by navigating to **{{ your username provided by Show }}** > **Preferences** then changing **Theme** to **Light**, clicking **Apply**, then refreshing your browser tab. In the screenshots below, the username is `Administrator`.
- The red boxed sections in the screenshots indicate where you need to interact with the browser window by clicking or typing something, and blue boxes are areas of interest.
- Comments and notes are *italicized*.

Module 0: Introduction

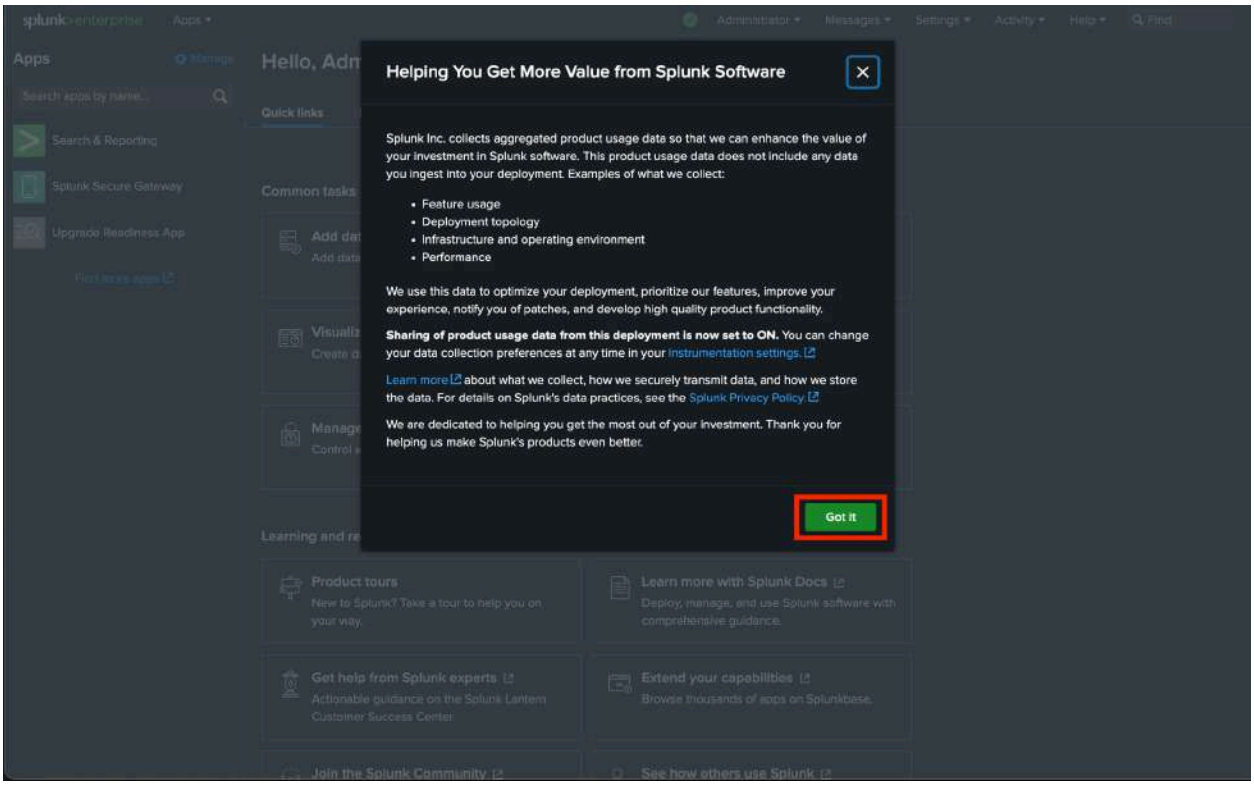
This module is designed to help you get acquainted with the environment that will be used during this workshop. You will be running a search to verify data is present in the environment, and running through one of the Splunk Machine Learning Toolkit (MLTK) Forecasting Examples in the MLTK Showcase. The MLTK Showcase is a great place to see what the MLTK can do, and find examples of machine learning workflows in Splunk.

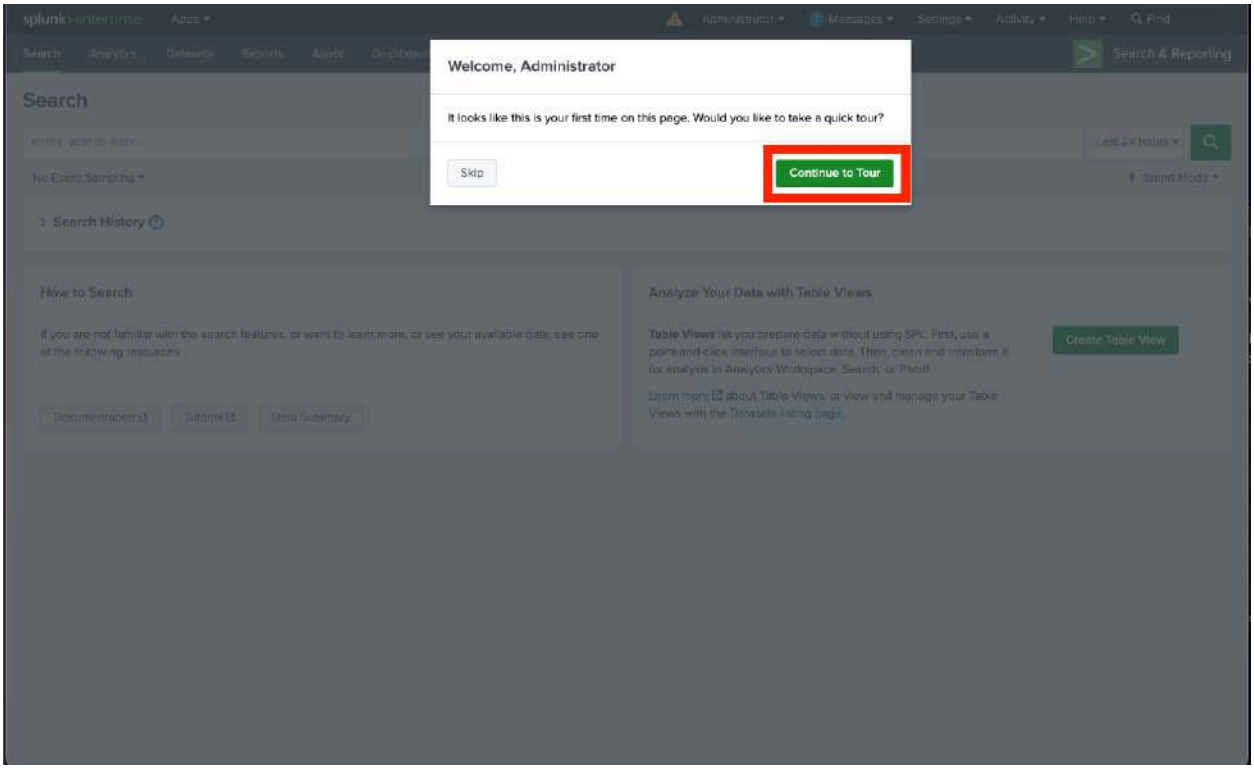
This module is not meant to be challenging and really just designed to make sure you can log into everything before the next module. If you run into problems or need help please ask!

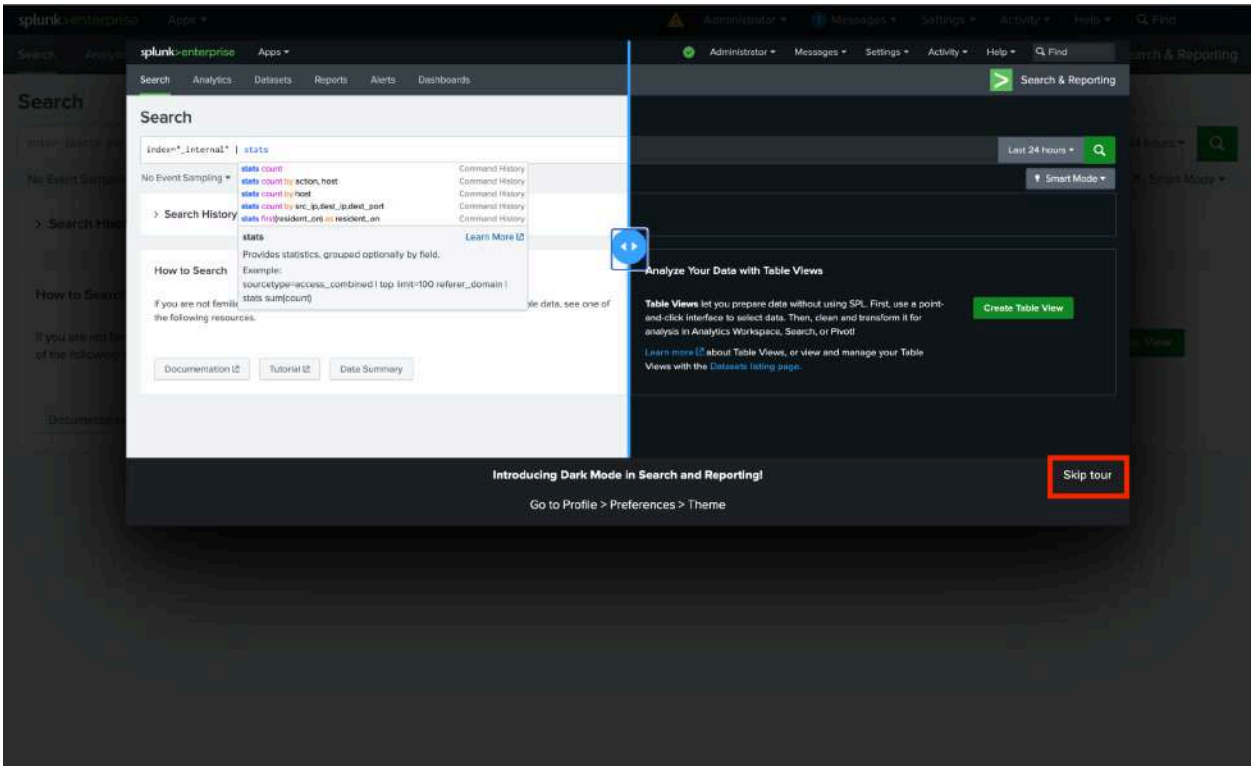
1. Open a browser tab and navigate to the Splunk Enterprise URL provided by Splunk Show. Log in using the provided `Username` and `Password`.



2. If you are prompted by a `Helping You Get More Value from Splunk Software`, `Welcome, user`, or `Introducing Dark Mode in Search and Reporting!` pop up, feel free to dismiss it by clicking **Skip**, **Skip tour**, or **Got it**.


Helping You Get More Value from Splunk Software
Splunk Inc. collects aggregated product usage data so that we can enhance the value of your investment in Splunk software. This product usage data does not include any data you ingest into your deployment. Examples of what we collect:
Feature usage
Deployment topology
Infrastructure and operating environment
Performance
We use this data to optimize your deployment, prioritize our features, improve your experience, notify you of patches, and develop high quality product functionality.
Sharing of product usage data from this deployment is now set to ON. You can change your data collection preferences at any time in your Instrumentation settings.
Learn more about what we collect, how we securely transmit data, and how we store the data. For details on Splunk's data practices, see the Splunk Privacy Policy.
We are dedicated to helping you get the most out of your investment. Thank you for helping us make Splunk's products even better.
Got it

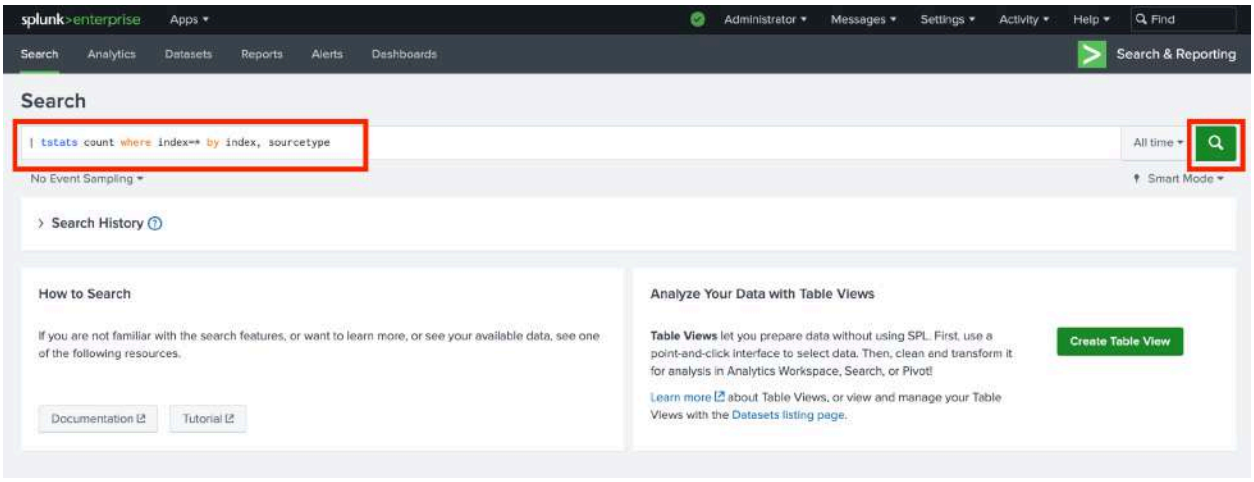

Welcome, Administrator
It looks like this is your first time on this page. Would you like to take a quick tour?
Skip
Continue to Tour



3. To discover what datasets are available in the Splunk instance, run the following search (copying and pasting it is easiest).

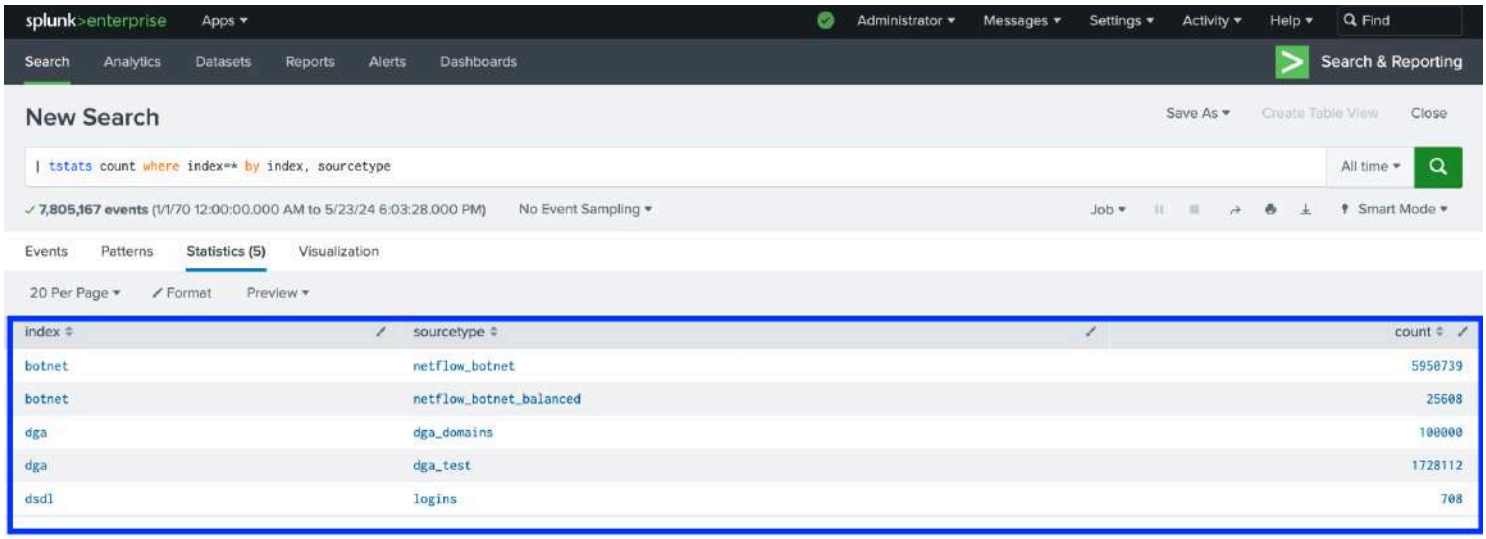
```
| tstats count where index=* by index, sourcetype
```

This search shows what data is available to be searched, split by index and sourcetype.

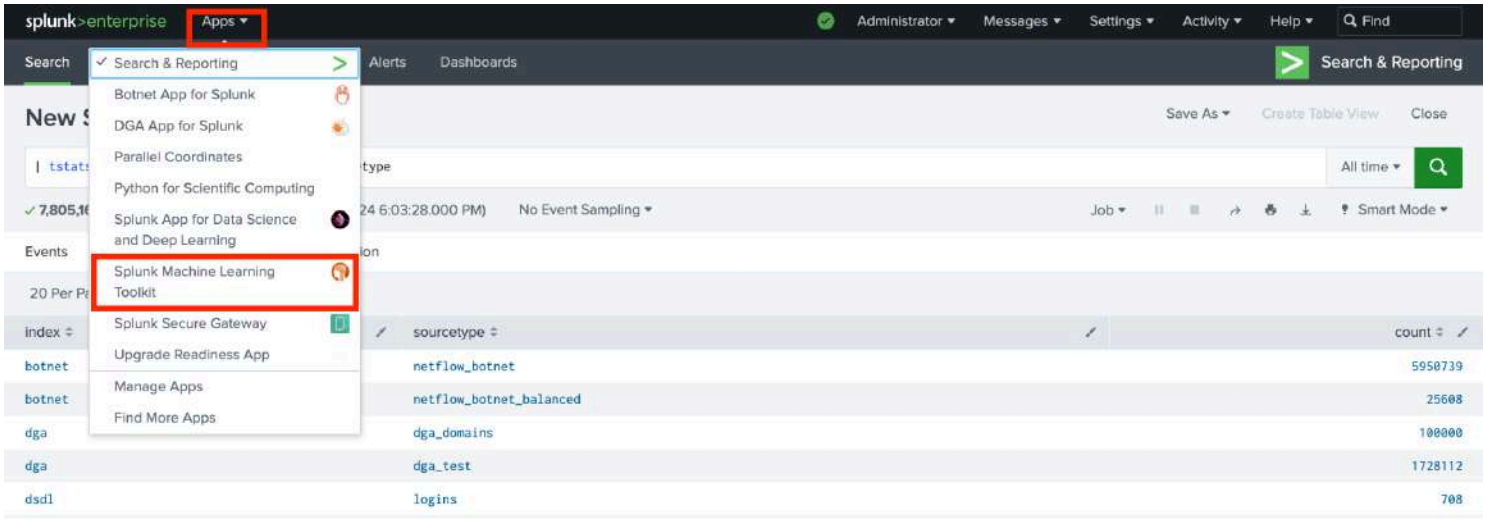


You should see five sourcetypes across three different indexes.

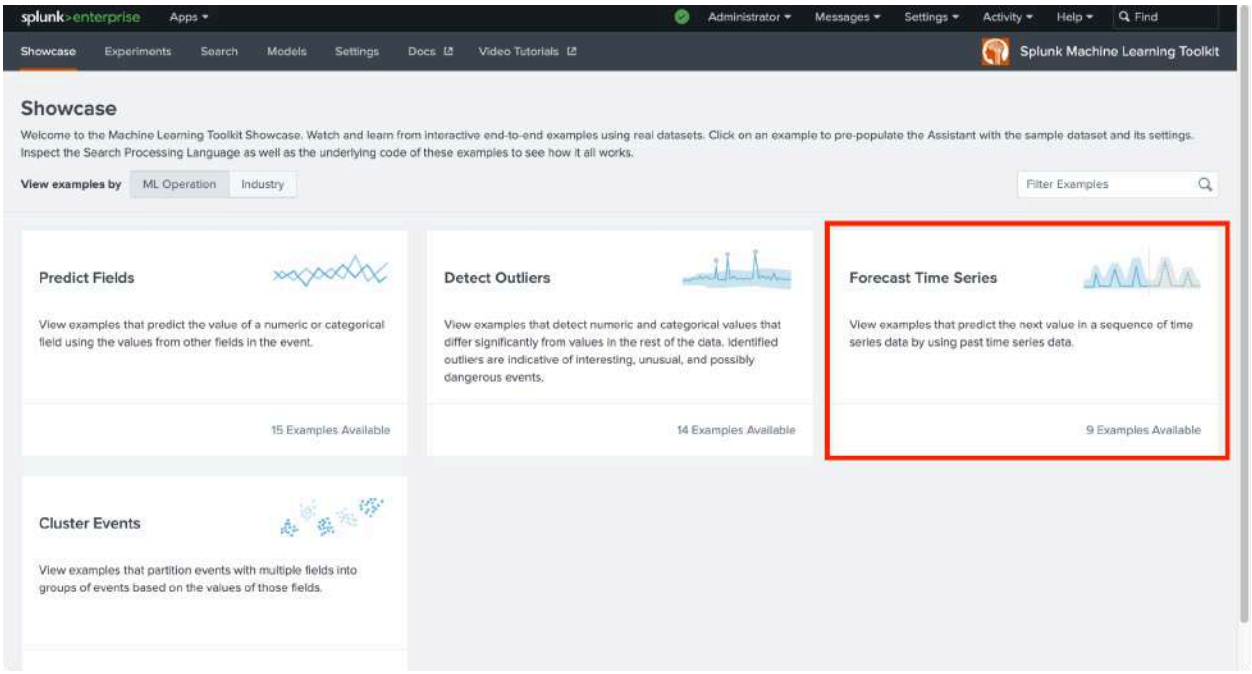
Please note that the time range picker is set to `All time`*. This was set intentionally and should not be changed during this workshop.*

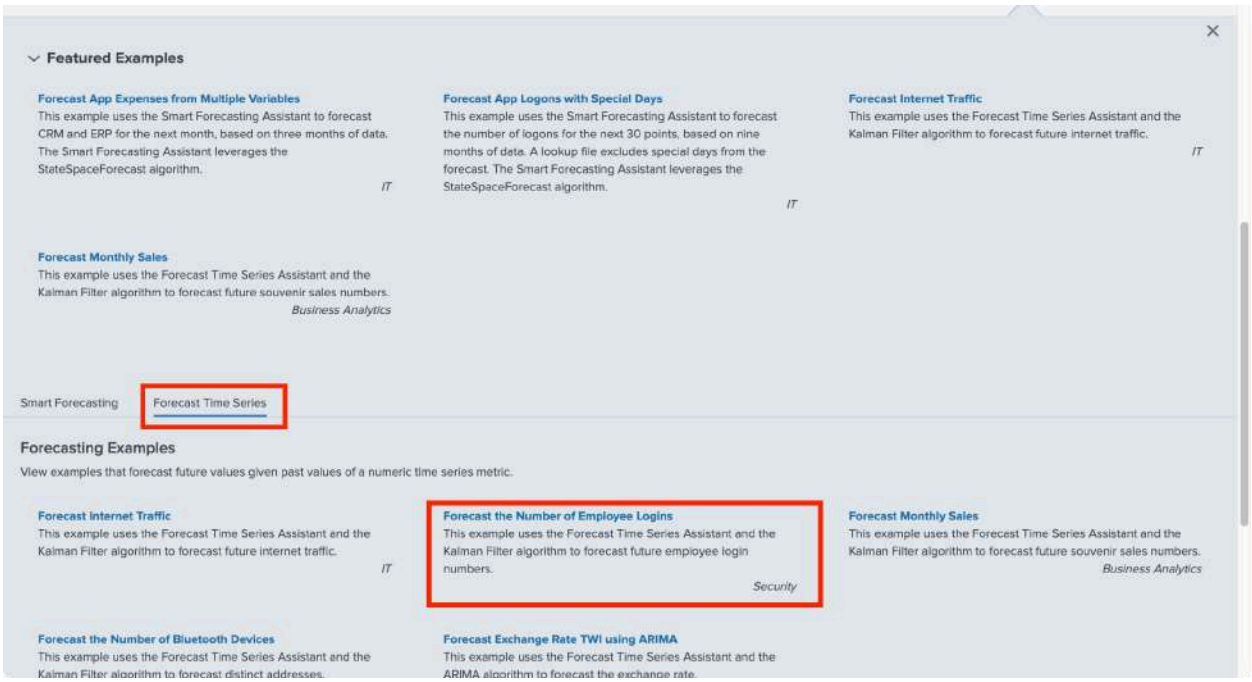


4. Navigate to the `Splunk Machine learning Toolkit` app by clicking **Apps** on the menu bar, then selecting **Splunk Machine Learning Toolkit**.

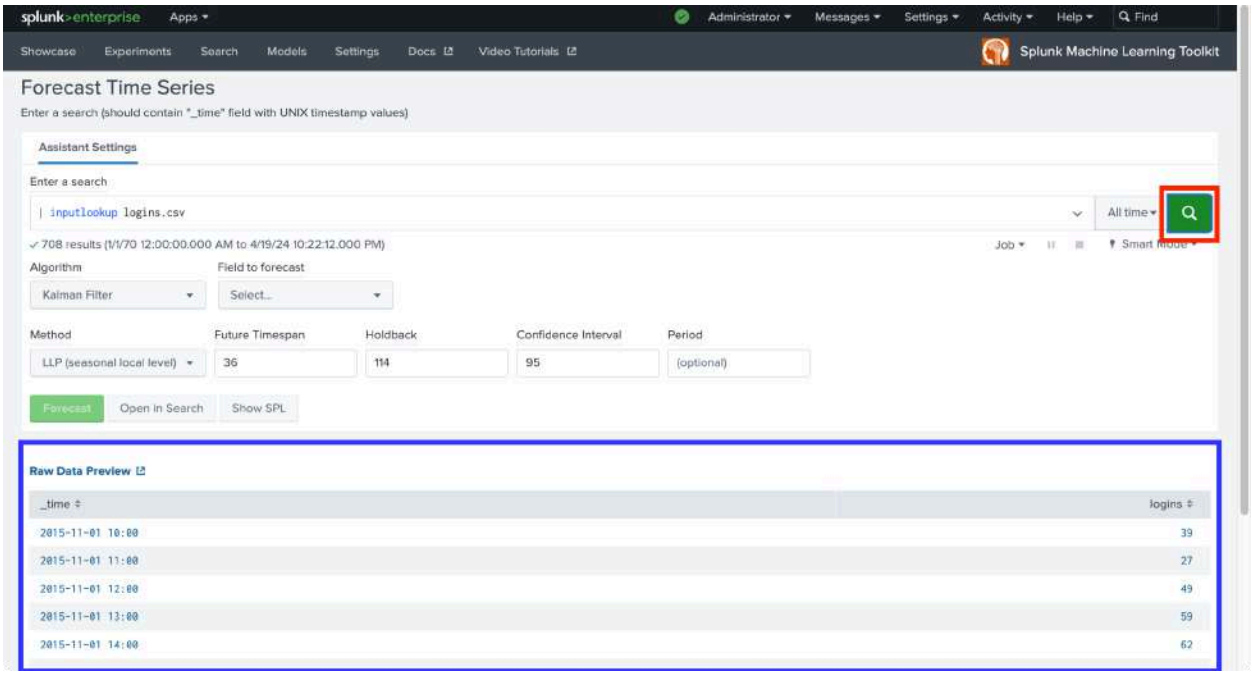


5. Click **Forecast Time Series**, select the **Forecast Time Series** tab (you may need to scroll down), then click the **Forecast the Number of Employee Logins** example.

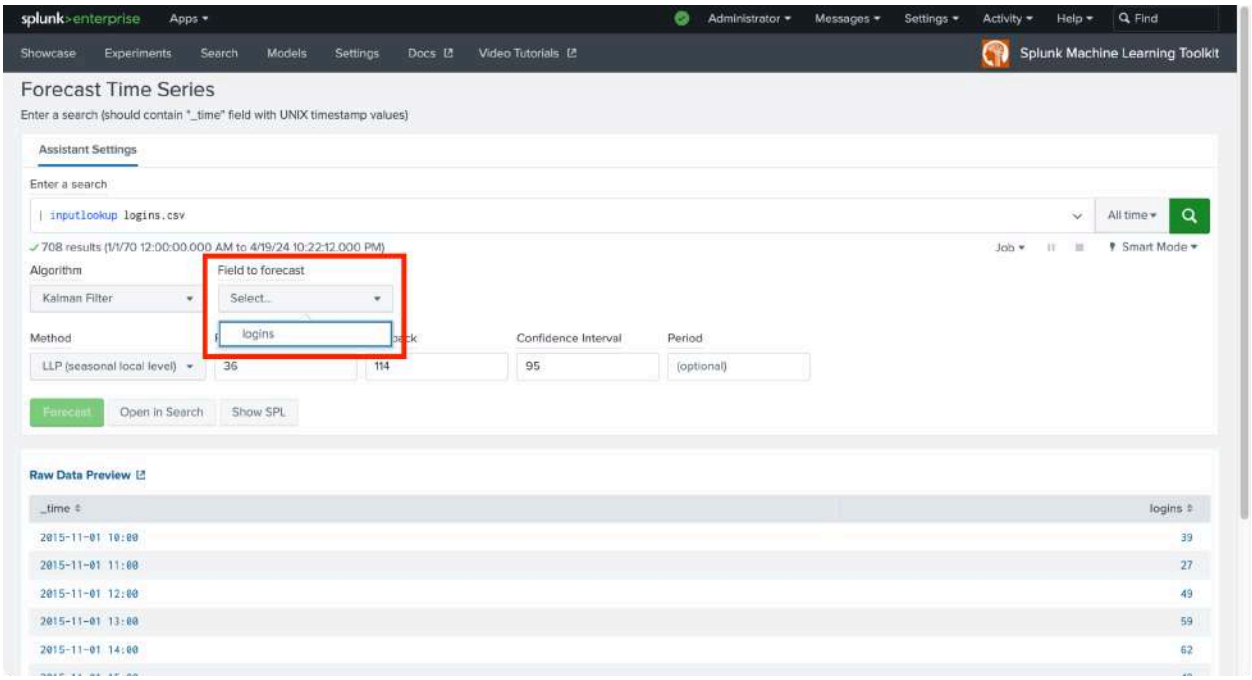




6. Click the search icon (magnifying glass) on the right to run a search that displays the training data.

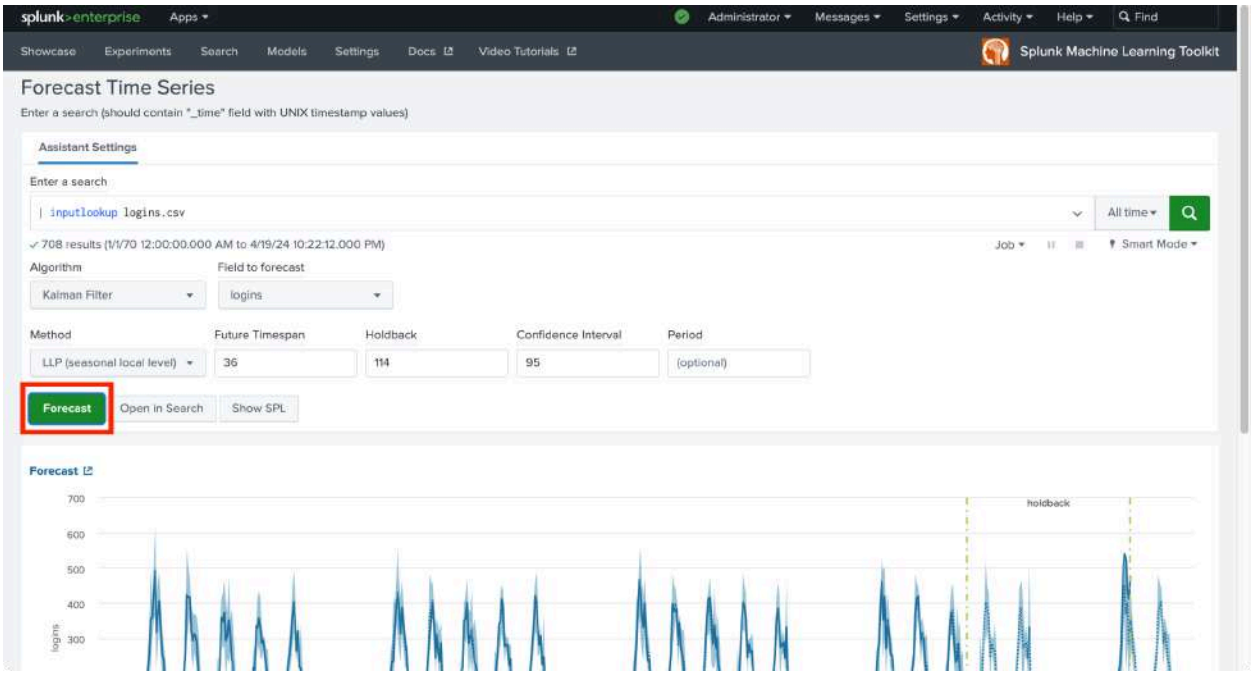


7. Click the **Field to forecast** dropdown, and select `logins`.

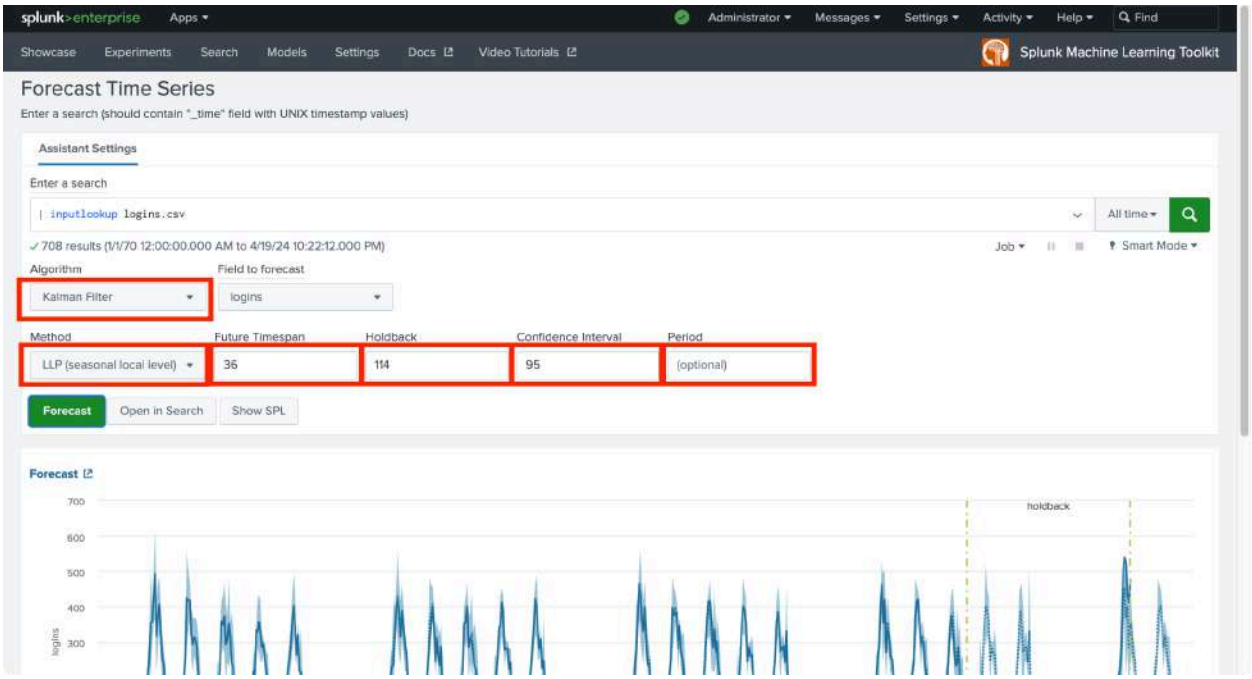


8. Click **Forecast** to start the forecast process, then view the results and accuracy under the **Forecast**. The forecast accuracy can be viewed under the R^2 `Statistic`, `Root Mean Squared Error (RMSE)`, and `Forecast Outliers` panels below the **Forecast** panel. For any of these panels, feel free to click **Show SPL** to view the SPL used to generate the dashboard panel shown.

   If you want to learn more about R^2 `Statistic`, `Root Mean Squared Error (RMSE)`, or `Forecast Outliers`, you can learn more by hovering above that text highlighted in blue.



9. Also, feel free to experiment by changing the forecast **Algorithm** and any of the hyperparameters such as **Method**, **Future Timespan**, **Holdback**, **Confidence Interval**, and **Period**.



That's it for Module 0! The real ML work will start in the next module.

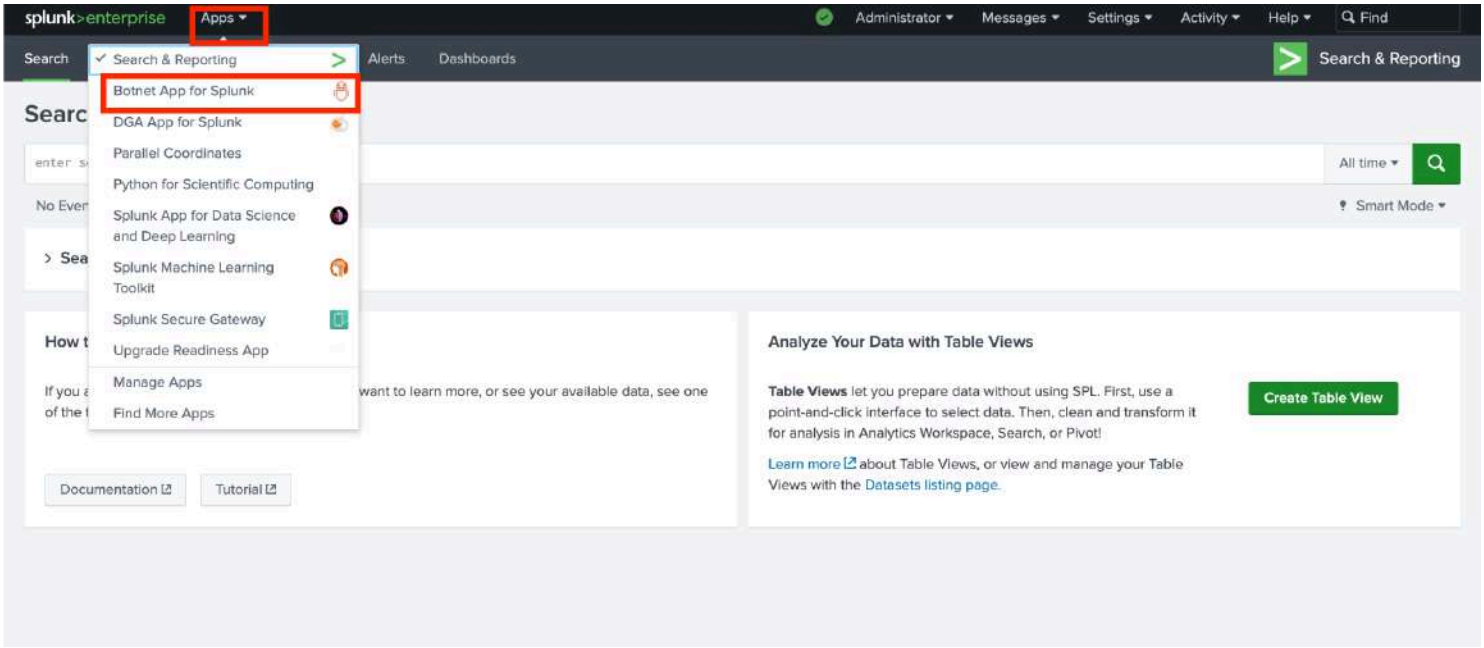
Module 1: Anomaly Detection

In this module, you will be working with refined netflow data to train anomaly detection models to identify traffic being generated by botnets. This module is loosely based on the Botnet App for Splunk in that it uses the same data sets and dashboards will be referenced during this module. This app is valuable to show what's possible with Splunk in regards to machine learning. However, this module will be deconstructing and walking through various parts of the app to understand how it works.

Task 1: Data Exploration

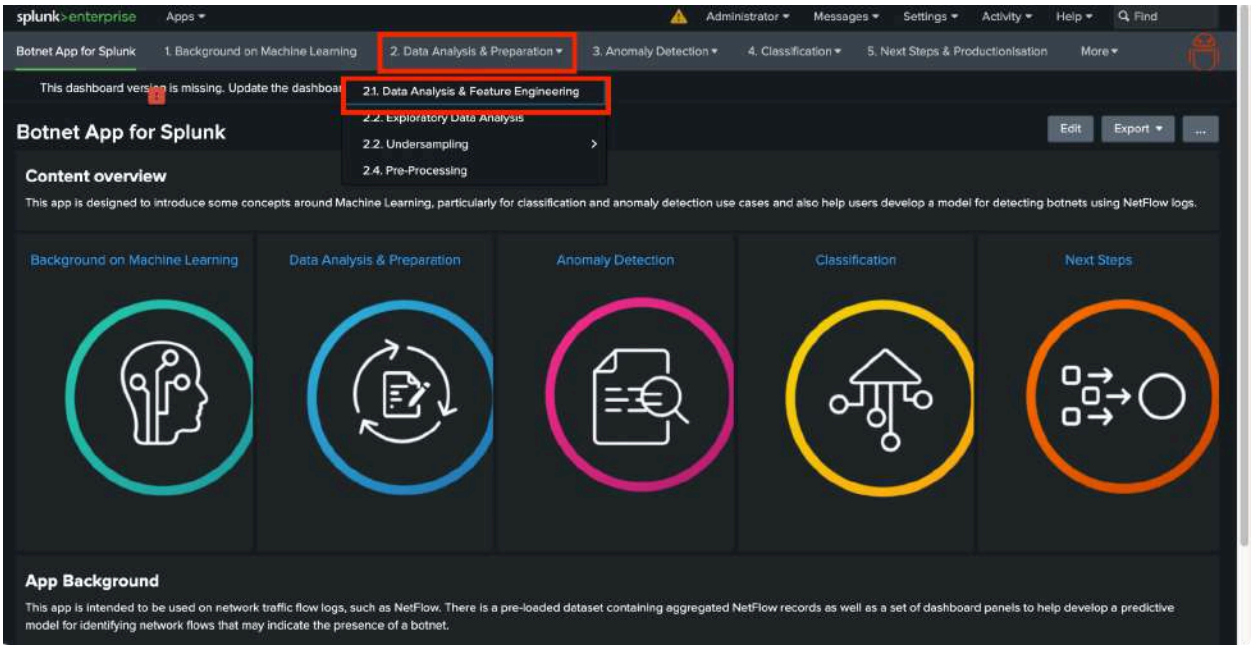
In this task, you will be exploring the data being used in this module. First, you'll take a look at the tables created in the Botnet App for Splunk, then create searches to explore the data in similar ways using the Search Processing Language (SPL). All of the data is in the `botnet` index, with the training dataset set as the `netflow_botnet` sourcetype, and the testing dataset set as the `netflow_botnet_balanced` sourcetype.

1. In the Splunk Enterprise host, navigate to the Botnet App for Splunk.

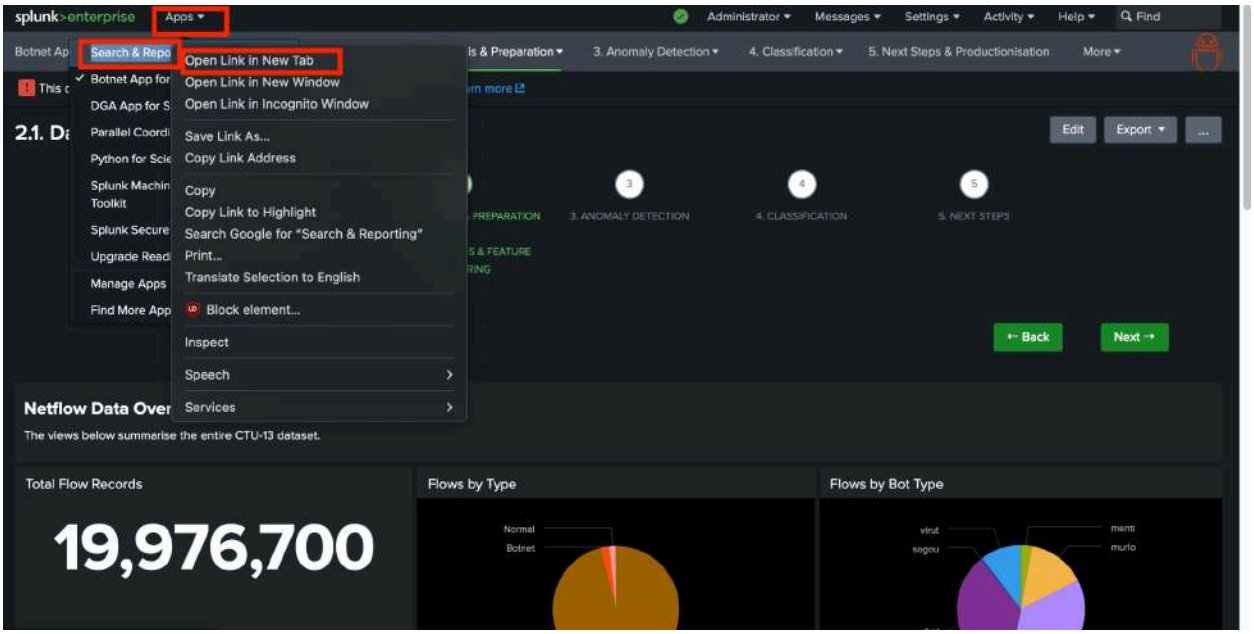


2. Expand the `2. Data Analysis & preparation` > `2.1 Data Analysis & Feature Engineering` dashboard.

   This panel might take a few minutes to load, and you do not need to wait for it to complete.

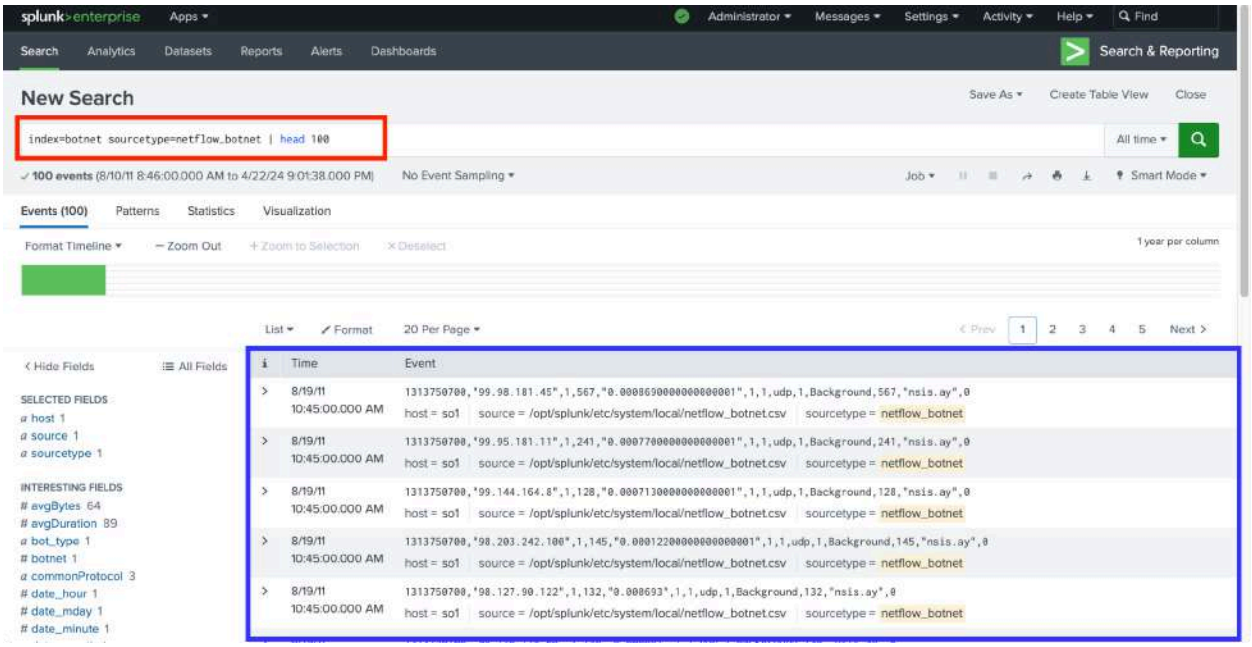


3. Open the `Search & Reporting` app in a new tab. You can do this by clicking **Apps**, right-clicking **Search & Reporting**, and clicking **Open link in New Tab**.



4. Run the search below to view a sample of the training data.

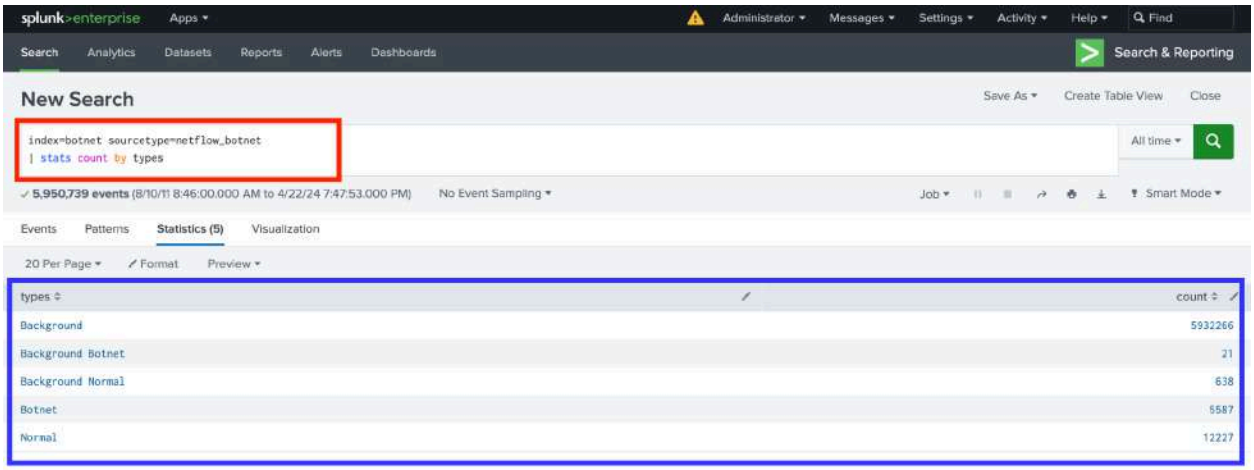
```
index=botnet sourcetype=netflow_botnet | head 100
```



5. Run the following search to view the count of events in the training dataset by traffic type. This is for the most part recreating the `Flows by Type` panel on the `2.1 Data Analysis & Feature Engineering` dashboard.

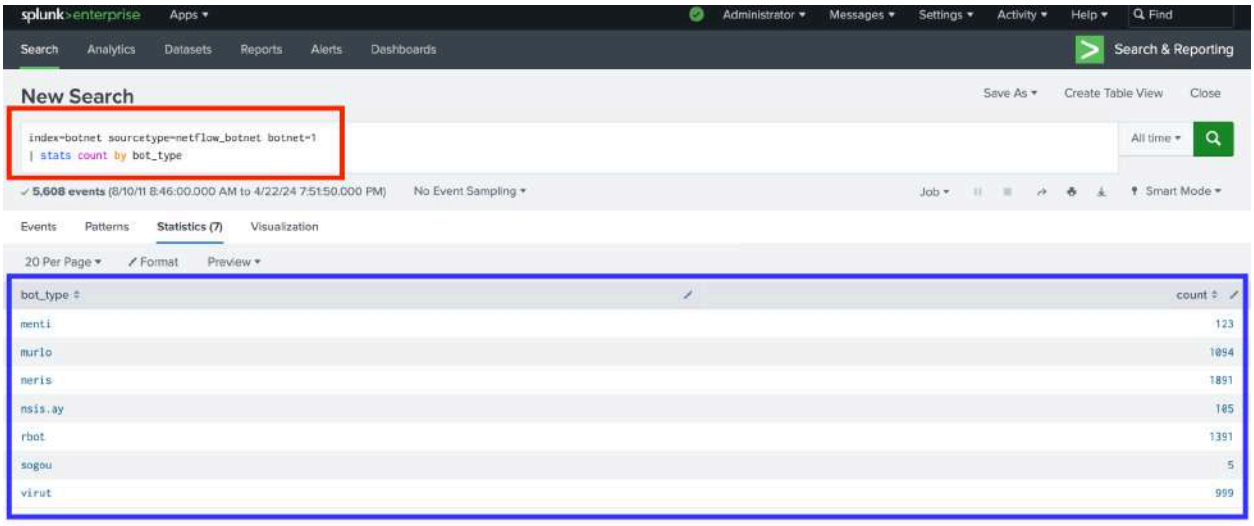
```
index=botnet sourcetype=netflow_botnet
| stats count by types
```

As you can see, most of the events originate from normal `Background` *traffic.*



6. Similarly, run the following search to view the count of events in the training dataset by botnet type. This is for the most part recreating the `Flows by Bot Type` panel on the `2.1 Data Analysis & Feature Engineering` dashboard.

```
index=botnet sourcetype=netflow_botnet botnet=1
| stats count by bot_type
```

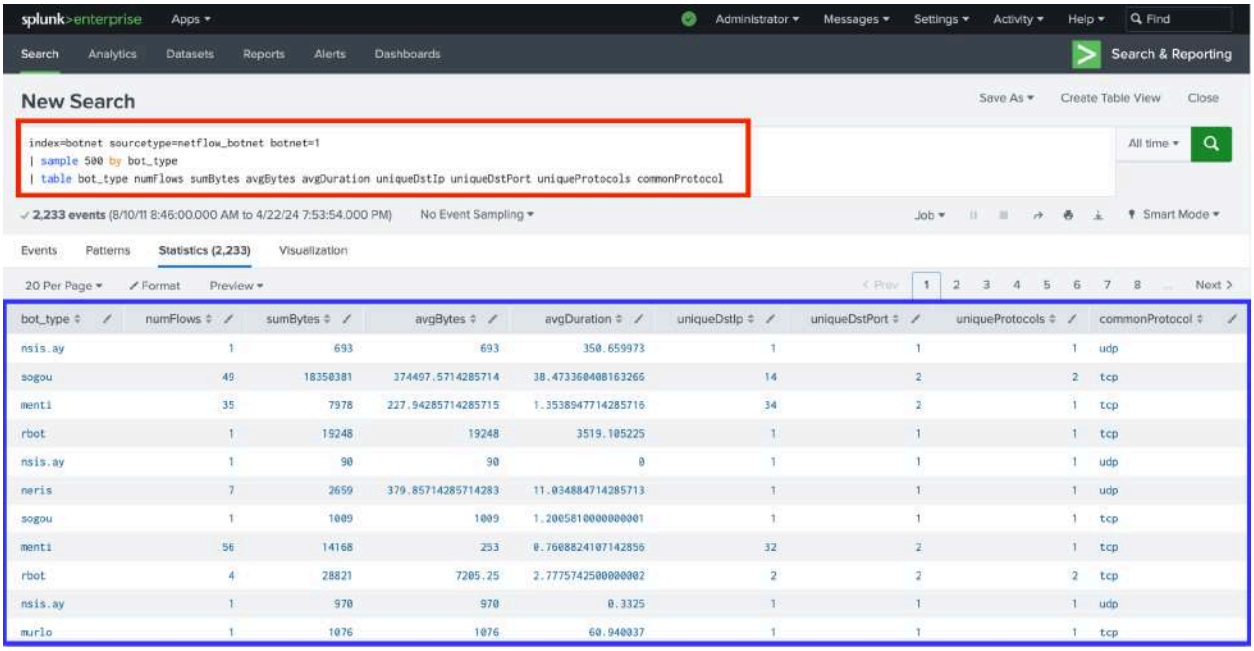


7. Run the search below to re-create the Feature Relationships on the `2.1 Data Analysis & Feature Engineering` dashboard, then click **Visualization**, and select the **Parallel Coordinates** visualization. This search and the related visualization are useful for identifying how different features relate to each other in the training data set.

   Note how the search is stratifying by botnet type by using the command `| sample 500 by bot_type`.

   Also, note how the features like bot_type and numFlows are related to each other.

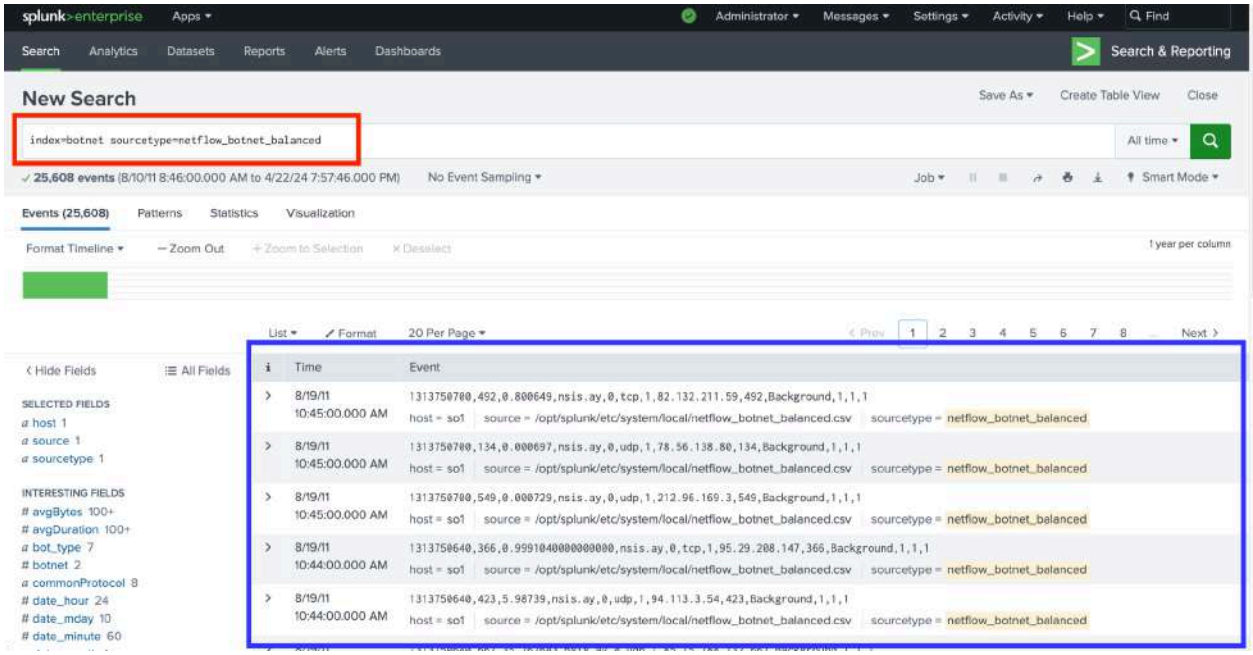
```
index=botnet sourcetype=netflow_botnet botnet=1
| sample 500 by bot_type
| table bot_type numFlows sumBytes avgBytes avgDuration uniqueDstIp uniqueDstPort
uniqueProtocols commonProtocol
```



splunk>

8. Run the following search to view the test data set.

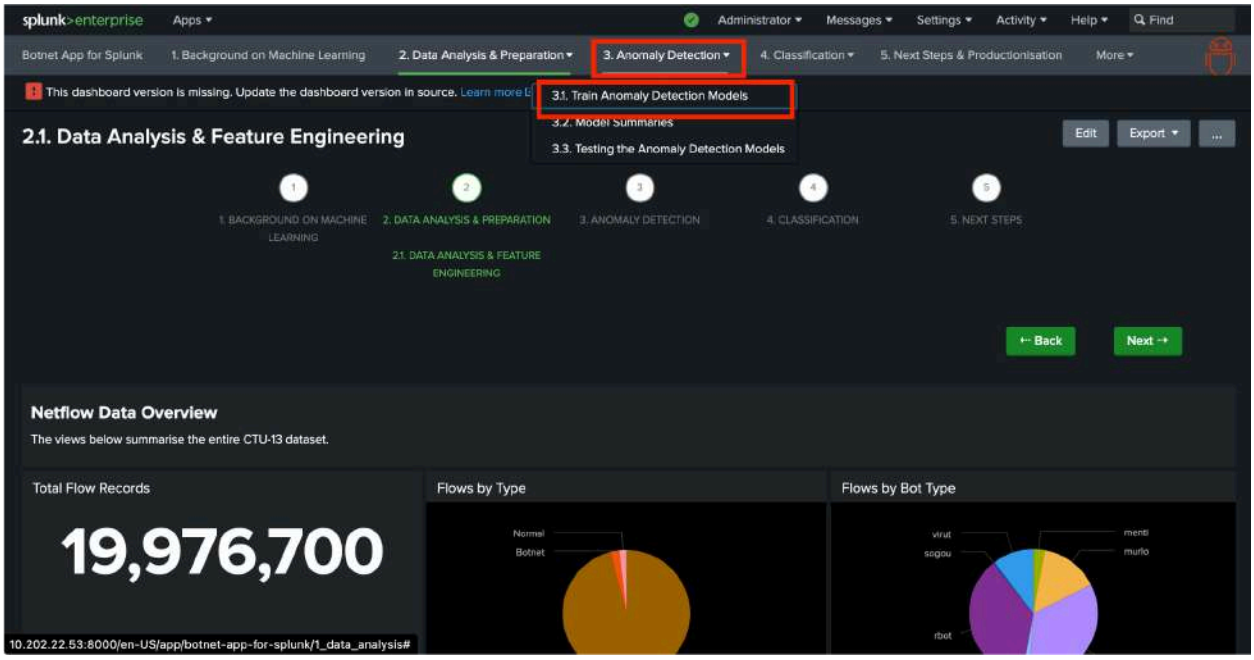
```
index=botnet sourcetype=netflow_botnet_balanced
```



Task 2: Model Training

The features in the testing and training datasets are already in a numeric format we can use, so feature engineering can be skipped! This means we can progress right to training the models. You will train multiple Density Function models referencing different features in the training data set, with the goal of using the trained models to identify botnet traffic (and by extension normal traffic because anything not anomalous for botnet traffic will be normal traffic).

1. In the browser tab open to the Botnet App for Splunk, expand the options under `3. Anomaly Detection`, and open the `3.1 Train Anomaly Detection Models` dashboard. In the app, this is the dashboard that is used to train the models.



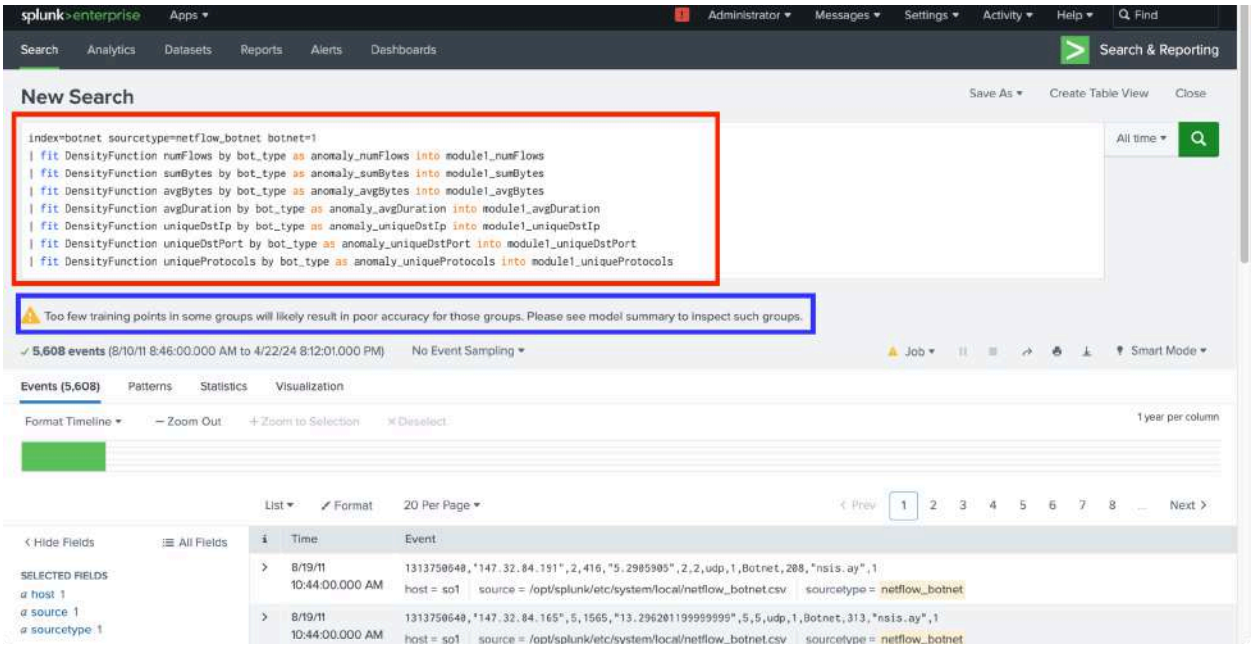
2. In the browser tab open to the `Search & Reporting` app, run the following search to train 7 new Density Function models, each trained to look for anomalies in different features.

   In our testing, this search took about 90 seconds to complete.

   Note how each `fit` command runs against a different feature from the dataset, this search is only looking for botnet data, and it is stratifying across botnet type.

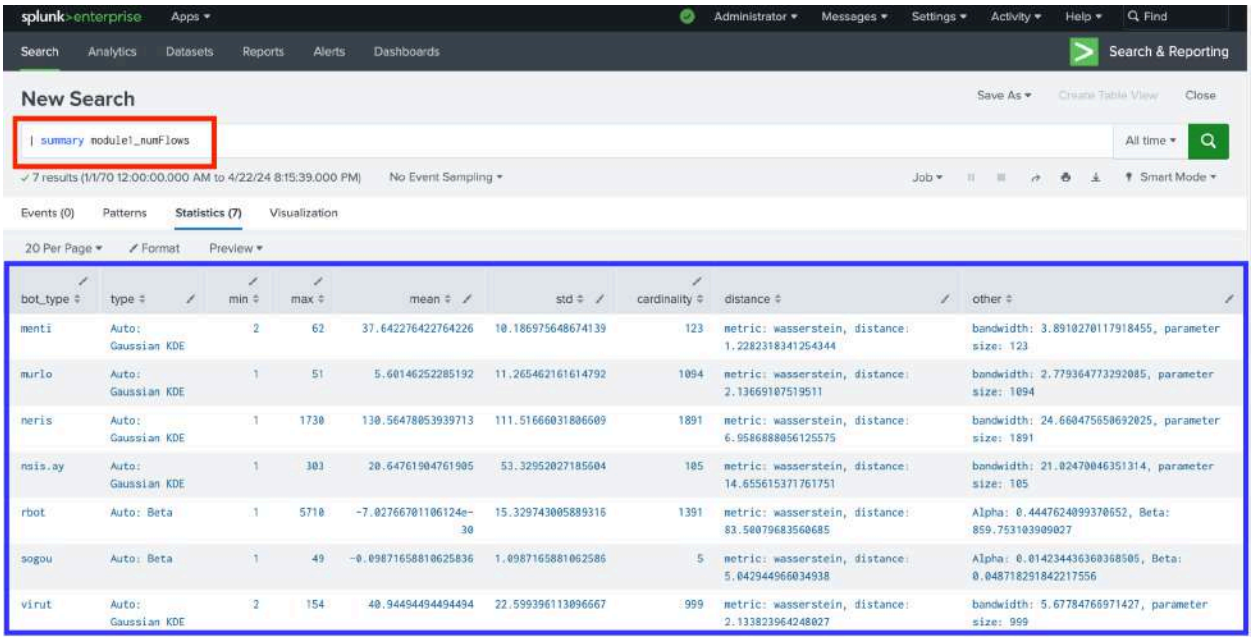
   Feel free to ignore any warnings about undersampling or having too few data sets.

```
index=botnet sourcetype=netflow_botnet botnet=1
| fit DensityFunction numFlows by bot_type as anomaly_numFlows into module1_numFlows
| fit DensityFunction sumBytes by bot_type as anomaly_sumBytes into module1_sumBytes
| fit DensityFunction avgBytes by bot_type as anomaly_avgBytes into module1_avgBytes
| fit DensityFunction avgDuration by bot_type as anomaly_avgDuration into
module1_avgDuration
| fit DensityFunction uniqueDstIp by bot_type as anomaly_uniqueDstIp into
module1_uniqueDstIp
| fit DensityFunction uniqueDstPort by bot_type as anomaly_uniqueDstPort into
module1_uniqueDstPort
| fit DensityFunction uniqueProtocols by bot_type as anomaly_uniqueProtocols into
module1_uniqueProtocols
```



3. You can view the specific model information for any trained models with the `summary` command. Run the following search to view the details of the module1_numFlows model. Feel free to repeat this throughout the workshop to view specific trained model parameters.

```
| summary module1_numFlows
```

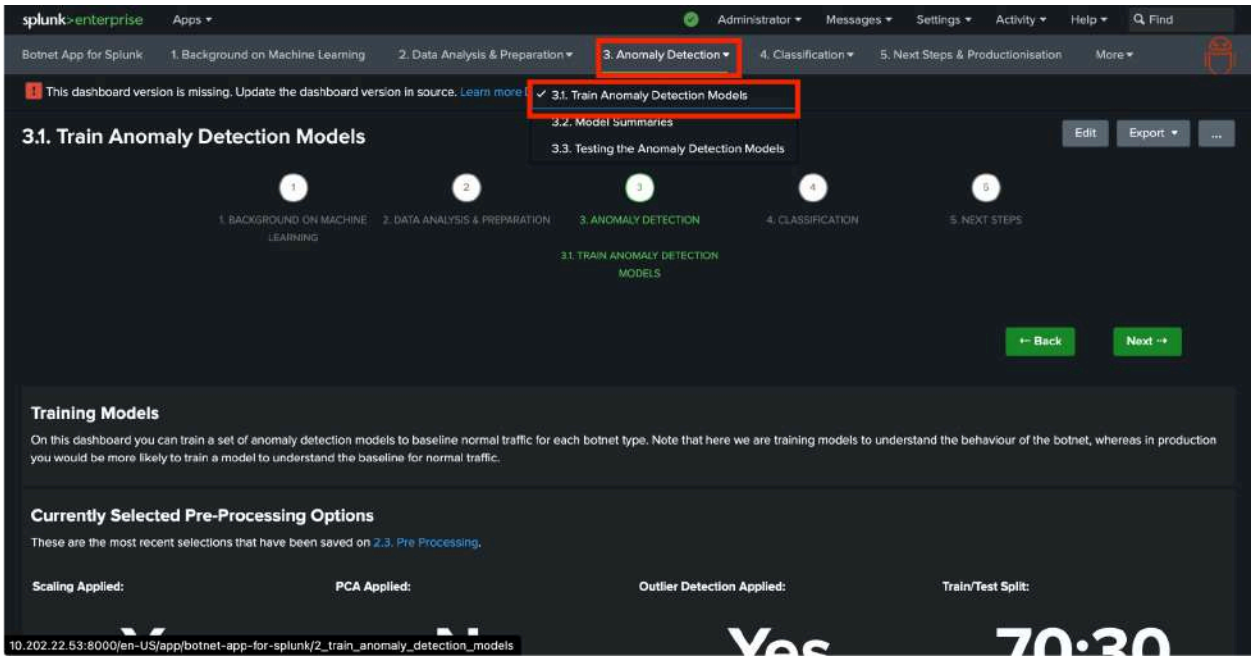


Task 3: Model Testing

Now that the models have been trained, it's time to test them to see how they perform against the test dataset! There is a dashboard in the Botnet App for Splunk to help with this, but you will be re-creating many of those panels in SPL.

1. In the browser tab open to the `Botnet App for Splunk`, expand the options under `3. Anomaly Detection`, and open the `3.3 Testing the Anomaly Detection Models` dashboard. In the app, this is the dashboard that is used to test the anomaly detection models.

   Note the confusion matrix spread across the last four panels.



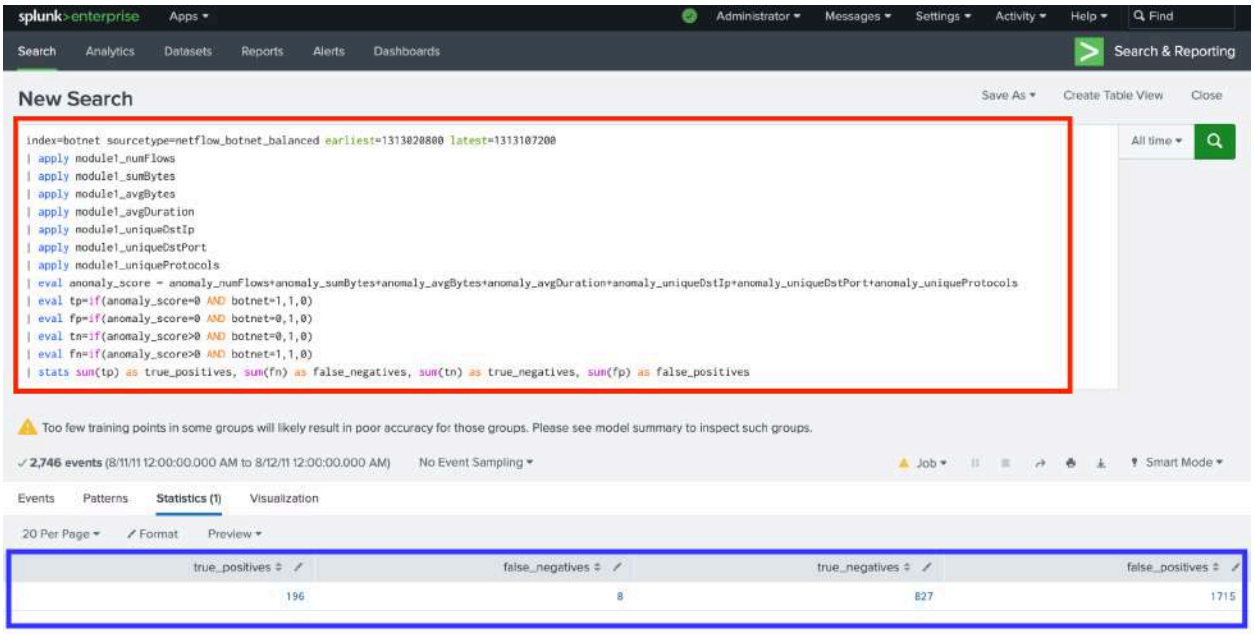
2. In the browser tab open to the `Search & Reporting` app, recreate the confusion matrix from the dashboard by running the following search.

   In our testing, this search took about 2 minutes to complete.

   Note how the search is scoped to a single day by setting the earliest and latest fields in the first search command. This was done to reduce the dataset size needing to run through all seven models, to reduce the time it takes for this search to run.

   The search works by trying to identify any anomalous network flows, then checking if those flows are from botnets or not. Keep in mind that these models are trained off of what "normal" looks like for a botnet, so anything identified as anomalous is most likely normal/non-botnet traffic.
   - *If the anomaly score is zero, and the traffic originates from a botnet, it is a true positive (correctly identified as coming from a botnet)*
   - *If the anomaly score is zero, and the traffic did not originate from a botnet, it is a false positive (incorrectly identified as coming from a botnet when it was normal traffic)*
   - *If the anomaly score is greater than zero, and the traffic did not originate from a botnet, it is a true negative (correctly identified as normal traffic)*
   - *If the anomaly score is greater than zero, and the traffic did originate from a botnet, it is a false negative (incorrectly identified as normal traffic but it originated from a botnet)*

```
index=botnet sourcetype=netflow_botnet_balanced earliest=1313020800 latest=1313107200
| apply module1_numFlows
| apply module1_sumBytes
| apply module1_avgBytes
| apply module1_avgDuration
| apply module1_uniqueDstIp
| apply module1_uniqueDstPort
| apply module1_uniqueProtocols
| eval anomaly_score =
anomaly_numFlows+anomaly_sumBytes+anomaly_avgBytes+anomaly_avgDuration+anomaly_unique
DstIp+anomaly_uniqueDstPort+anomaly_uniqueProtocols
| eval tp=if(anomaly_score=0 AND botnet=1,1,0)
| eval fp=if(anomaly_score=0 AND botnet=0,1,0)
| eval tn=if(anomaly_score>0 AND botnet=0,1,0)
| eval fn=if(anomaly_score>0 AND botnet=1,1,0)
| stats sum(tp) as true_positives, sum(fn) as false_negatives, sum(tn) as
true_negatives, sum(fp) as false_positives
```



3. Those scores aren't great, but what happens if a custom threshold parameter is defined (default is 0.01)? Run the following two searches; the first one having a threshold of 0.05, and the second a parameter of 0.1.

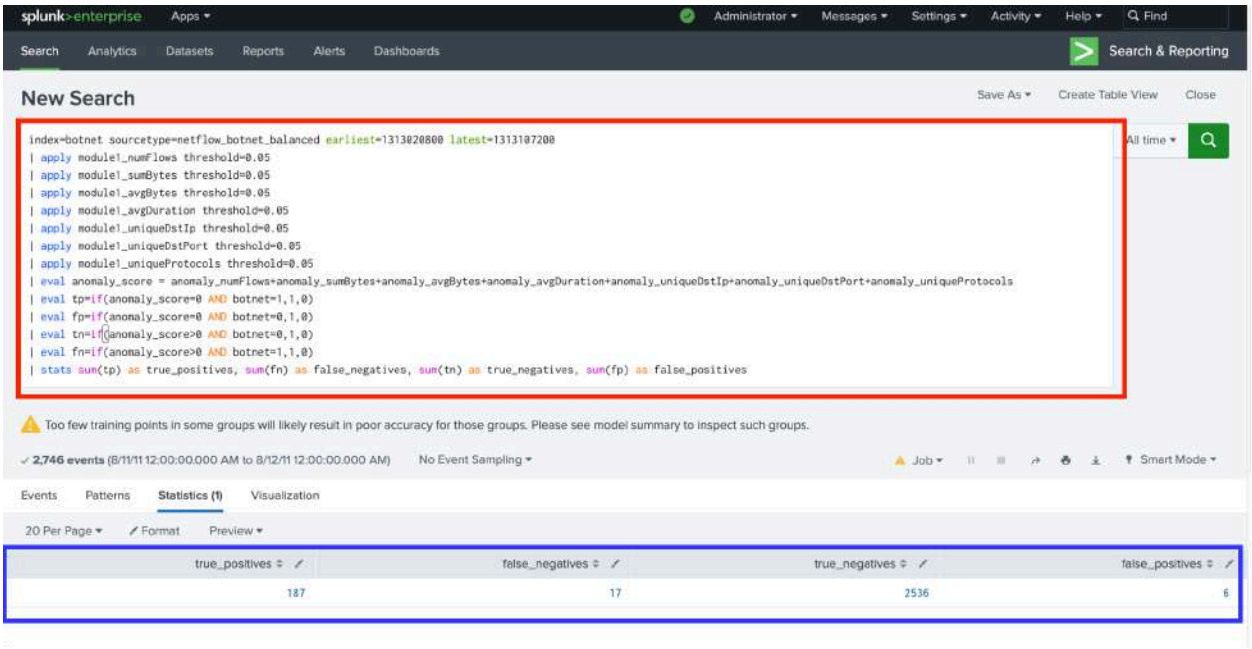
   Just another note that each of these searches took about 2 minutes to run while testing this workshop.

```
index=botnet sourcetype=netflow_botnet_balanced earliest=1313020800 latest=1313107200
| apply module1_numFlows threshold=0.05
| apply module1_sumBytes threshold=0.05
| apply module1_avgBytes threshold=0.05
| apply module1_avgDuration threshold=0.05
| apply module1_uniqueDstIp threshold=0.05
| apply module1_uniqueDstPort threshold=0.05
| apply module1_uniqueProtocols threshold=0.05
| eval anomaly_score =
anomaly_numFlows+anomaly_sumBytes+anomaly_avgBytes+anomaly_avgDuration+anomaly_unique
DstIp+anomaly_uniqueDstPort+anomaly_uniqueProtocols
| eval tp=if(anomaly_score=0 AND botnet=1,1,0)
| eval fp=if(anomaly_score=0 AND botnet=0,1,0)
| eval tn=if(anomaly_score>0 AND botnet=0,1,0)
| eval fn=if(anomaly_score>0 AND botnet=1,1,0)
| stats sum(tp) as true_positives, sum(fn) as false_negatives, sum(tn) as
true_negatives, sum(fp) as false_positives
```

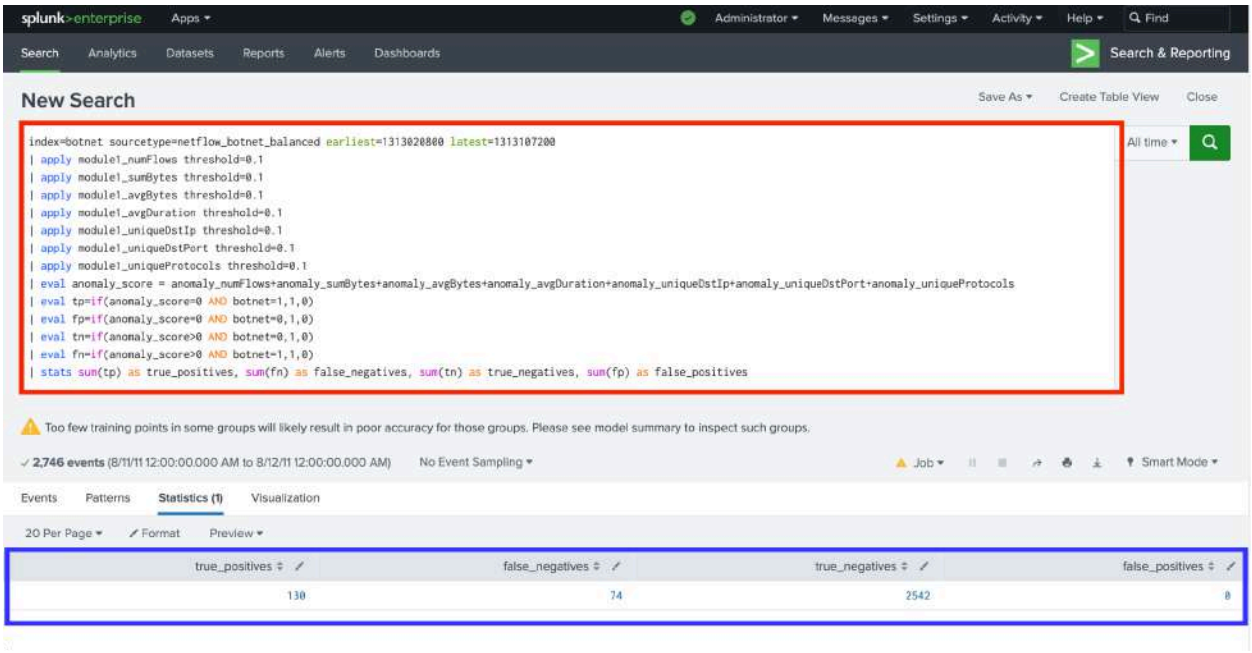
```
index=botnet sourcetype=netflow_botnet_balanced earliest=1313020800 latest=1313107200
| apply module1_numFlows threshold=0.1
| apply module1_sumBytes threshold=0.1
| apply module1_avgBytes threshold=0.1
```

```
| apply module1_avgDuration threshold=0.1
| apply module1_uniqueDstIp threshold=0.1
| apply module1_uniqueDstPort threshold=0.1
| apply module1_uniqueProtocols threshold=0.1
| eval anomaly_score =
anomaly_numFlows+anomaly_sumBytes+anomaly_avgBytes+anomaly_avgDuration+anomaly_unique
DstIp+anomaly_uniqueDstPort+anomaly_uniqueProtocols
| eval tp=if(anomaly_score=0 AND botnet=1,1,0)
| eval fp=if(anomaly_score=0 AND botnet=0,1,0)
| eval tn=if(anomaly_score>0 AND botnet=0,1,0)
| eval fn=if(anomaly_score>0 AND botnet=1,1,0)
| stats sum(tp) as true_positives, sum(fn) as false_negatives, sum(tn) as
true_negatives, sum(fp) as false_positives
```

Note how much better the results look in the confusion matrix! Most of the time it's worth tuning and testing different hyperparameter settings because it can greatly improve model accuracy.



splunk>



Task 4: Model Application

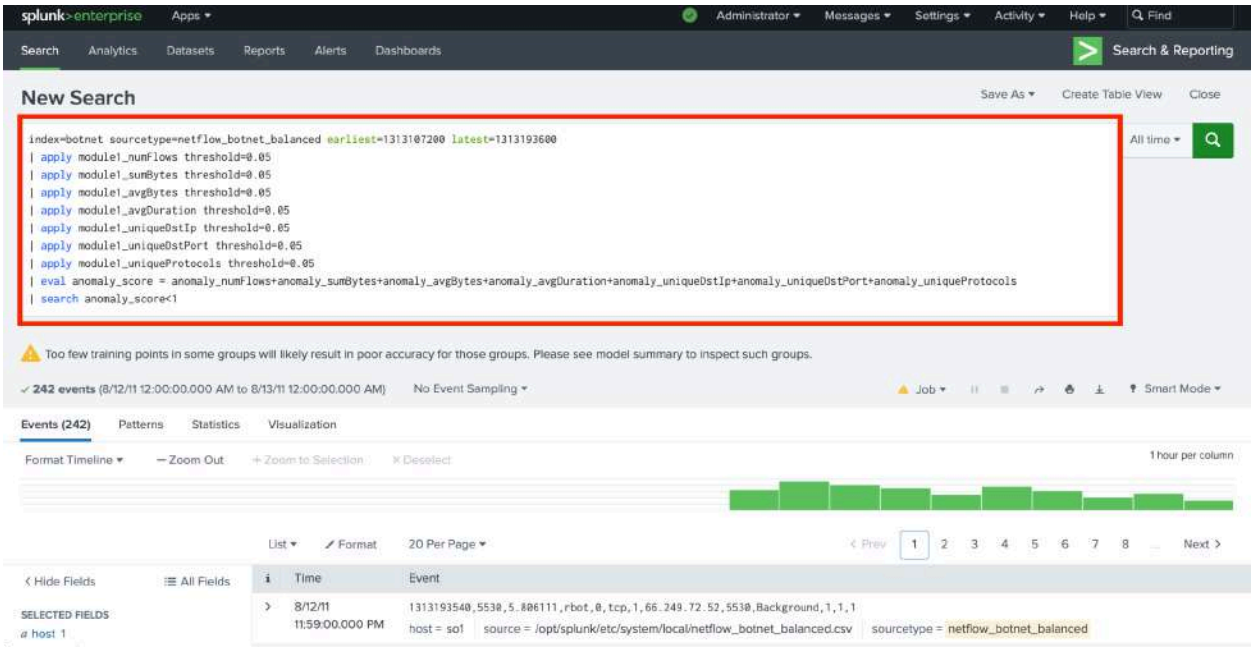
Now that the models are performing well, it's time to apply them.

1. An example search showing how the models can be applied is below. This search can be used to create an alert, or even an ES correlation search to identify hosts potentially communicating with botnets based on the observed network traffic. Any event that has an anomaly score below 1 is likely traffic originating from botnets.

   Notice again how the time range is scoped down to 1 day using the earliest and latest fields in the first search command. Again, this was just done to reduce the time workshop attendees would be sitting and waiting for searches to finish.

   It's worth noting again that since the anomaly detection models are trained to specifically identify botnet traffic, any traffic that is not anomalous is likely to be botnet traffic.

```
index=botnet sourcetype=netflow_botnet_balanced earliest=1313107200 latest=1313193600
| apply module1_numFlows threshold=0.05
| apply module1_sumBytes threshold=0.05
| apply module1_avgBytes threshold=0.05
| apply module1_avgDuration threshold=0.05
| apply module1_uniqueDstIp threshold=0.05
| apply module1_uniqueDstPort threshold=0.05
| apply module1_uniqueProtocols threshold=0.05
| eval anomaly_score =
anomaly_numFlows+anomaly_sumBytes+anomaly_avgBytes+anomaly_avgDuration+anomaly_unique
DstIp+anomaly_uniqueDstPort+anomaly_uniqueProtocols
| search anomaly_score<1
```



That's all for Module 1! In this module, you learned how to train machine learning models to identify botnet and normal traffic.

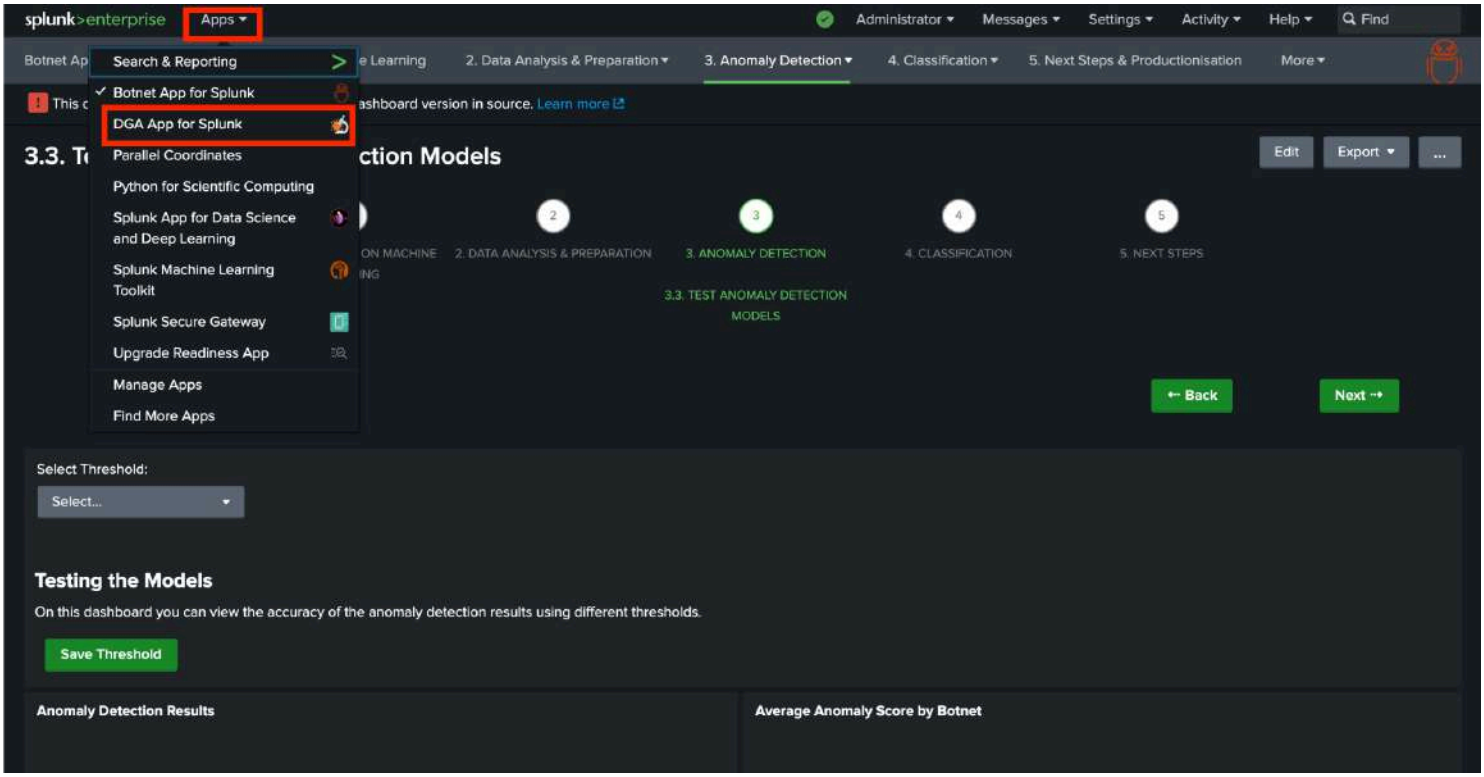
Module 2: Clustering

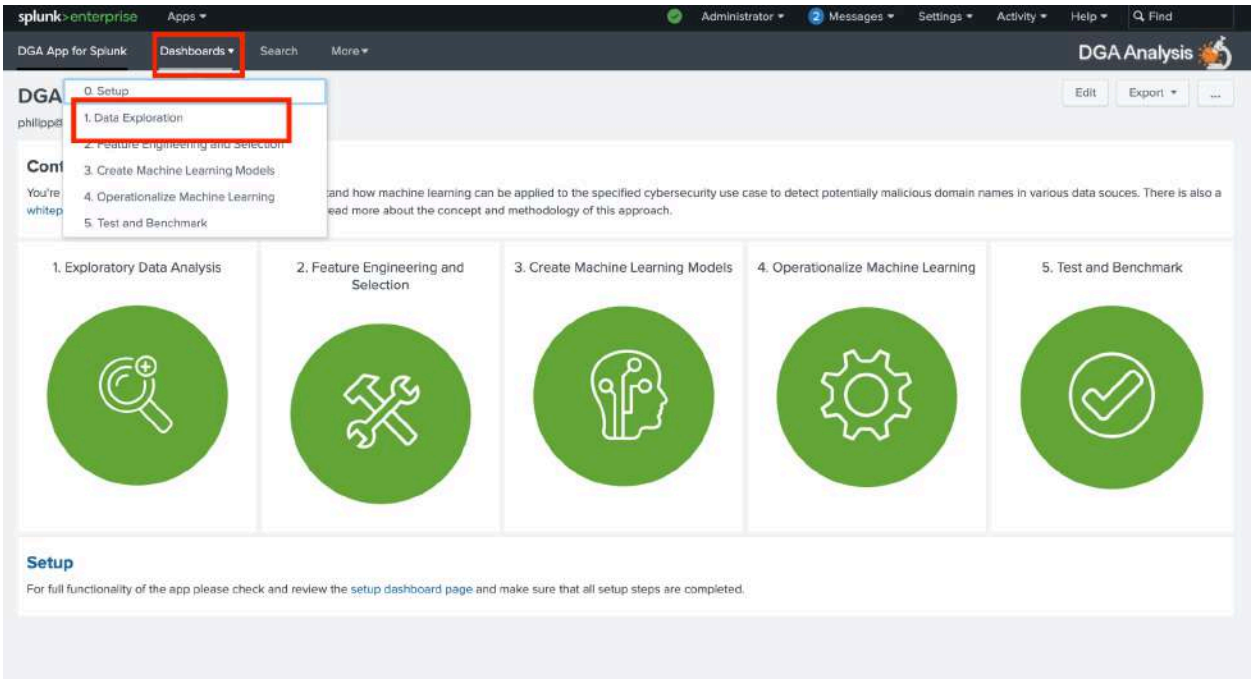
In this module, you will be working with domain names that are being communicated with from hosts in a network. Many of these domains have been generated by algorithms (aka Domain Generating/Generated Algorithms or DGAs). This module is loosely based on the DGA App for Splunk in that it uses data sets originating from and some of the dashboards from that app. Like the Botnet App for Splunk from Module 1, this app is valuable to show what is possible in regards to machine learning with Splunk. The goal of this module is to conduct feature engineering and exploration, then unsupervised learning with clustering.

Task 1: Data Exploration

In this task, you will be exploring the data that will be used in this module and Module 3. First, you will look at the raw data, then the results of N-gram analysis conducted in the DGA App for Splunk. After that you'll re-create the N-gram analysis in SPL. All of the data is in the `dga` index, with the training dataset set as the `dga_domains` sourcetype, and the testing dataset (used in Module 3) set as the `dga_test` sourcetype. This dataset is nearly identical to the dataset referenced in the DGA App for Splunk, but a semi-randomly generated timestamp has been added to each event from the first week of April 2024.

1. In the browser tab open to the Botnet App for Splunk from Module 1, navigate to the **DGA App for Splunk**, then expand the **Dashboards** dropdown and select the **1. Data Exploration** dashboard.

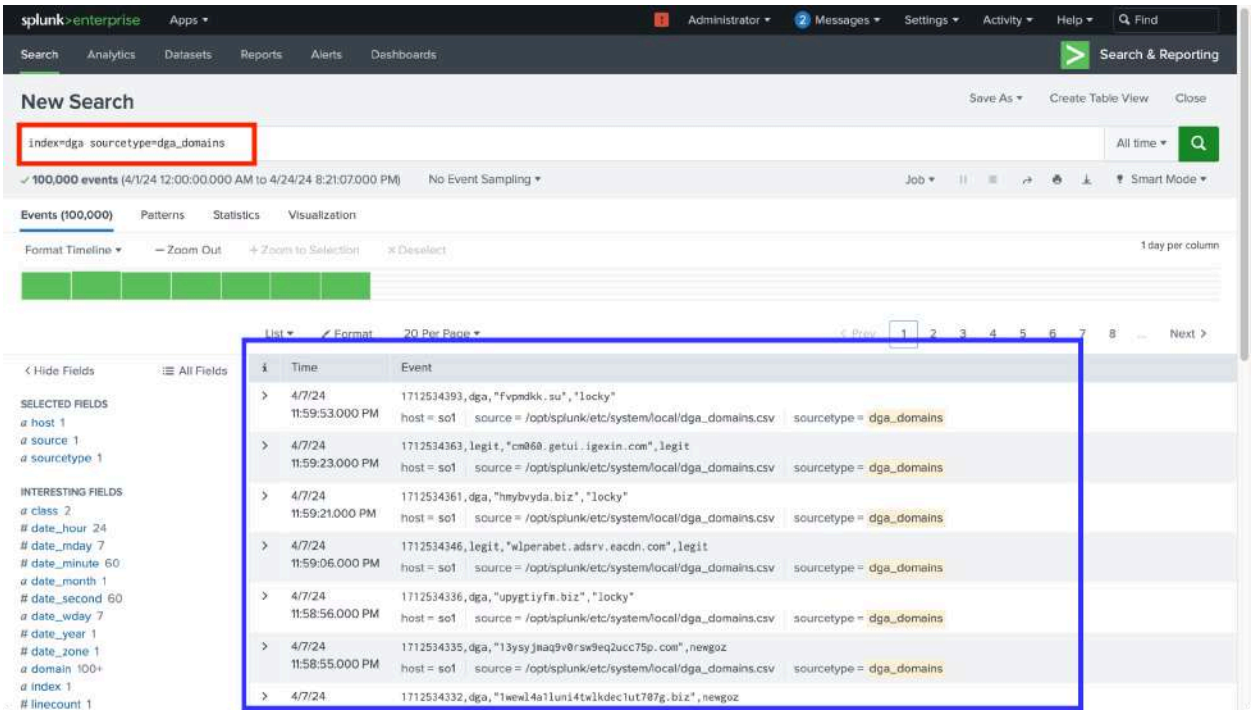




2. In the browser tab open to the `Search & Reporting` app, run the following search to show the raw data that this module and the next one will be referencing.

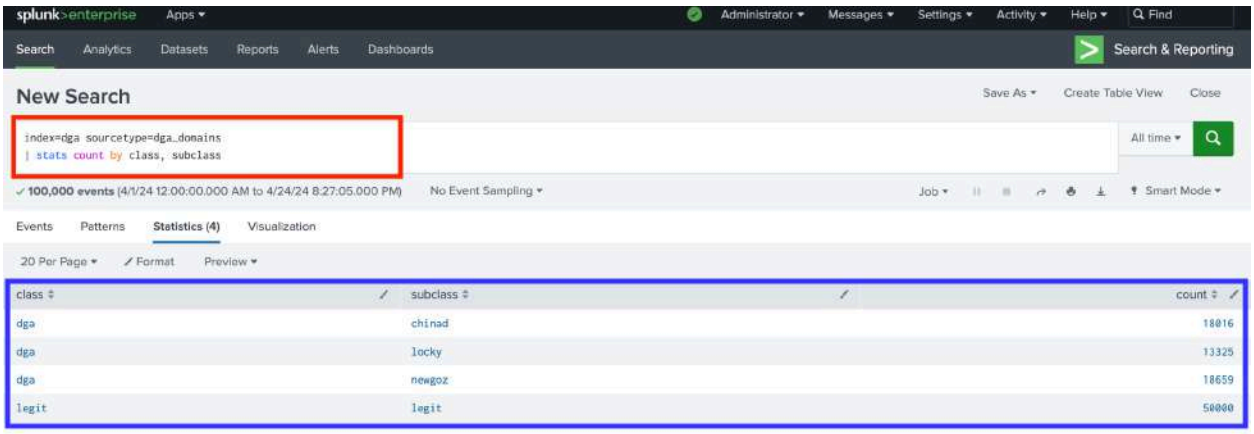
   ```
   index=dga sourcetype=dga_domains
   ```

   Note that each event consists of a timestamp, a domain, and is labeled with information related to whether the domain is a legitimate domain, or one generated by an algorithm. If it is generated with an algorithm, the event also specifies which algorithm generated the data.



3. Run the following search to view the count of events stratified by class (whether the domain is algorithmically-generated or legitimate) and subclass (which DGA created the domain if it is algorithmically-generated). This is similar to the upper-right panel on the `Dataset Overview` panel in the `DGA App for Splunk`'s `1. Data Exploration` dashboard.

```
index=dga sourcetype=dga_domains
| stats count by class, subclass
```



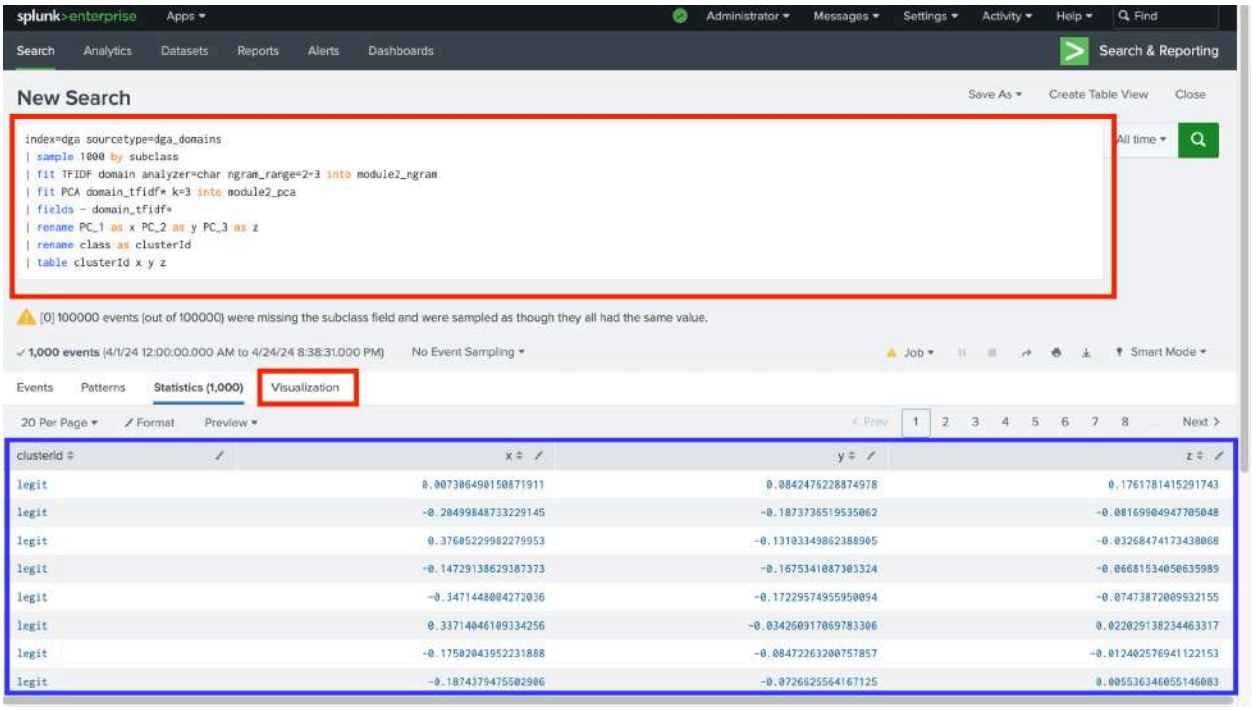
4. Run the search below to conduct an N-gram analysis of the domains, stratified by subclass, sampling by subclass, and (after visualizing the results) coloring based on class. After the search runs, graph the data in the **3D Scatterplot** visualization. This visualization is replicating the `n-gram analysis (2-3 char groups) of domain names with PCA k=3 by class` panel in the `DGA App for Splunk`.

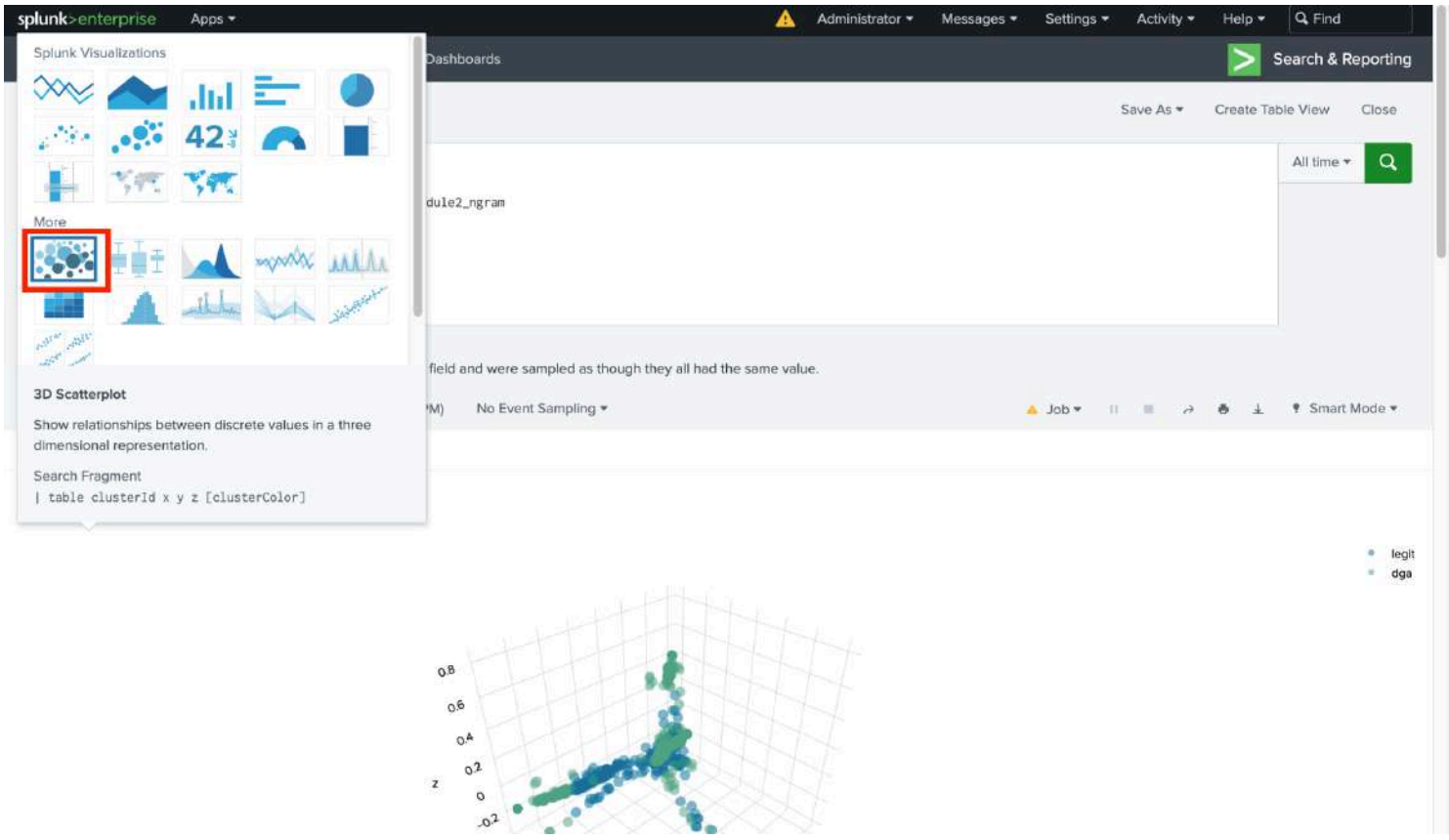
```
index=dga sourcetype=dga_domains
| sample 1000 by subclass
| fit TFIDF domain analyzer=char ngram_range=2-3 into module2_ngram
| fit PCA domain_tfidf* k=3 into module2_pca
| fields - domain_tfidf*
| rename PC_1 as x PC_2 as y PC_3 as z
| rename class as clusterId
| table clusterId x y z
```

This search is using TFIDF (the Splunk MLTK's implementation of scikit-learn's TfidfVectorizer algorithm) to create new features in the data, then using PCA for dimensionality reduction. The reduced dimensions are then set as the X, Y, Z, coordinates on the visualization.

You can click and rotate the graph to get a better view of the data. You can also scroll in and out of the visualization to zoom in and out. Just keep in mind that if you click the graph it will re-run the search with a filter for that specific area you clicked, and might change the results.

Note how there are clusters of domains with different colors, representing clusters of both legitimate and DGA domains.

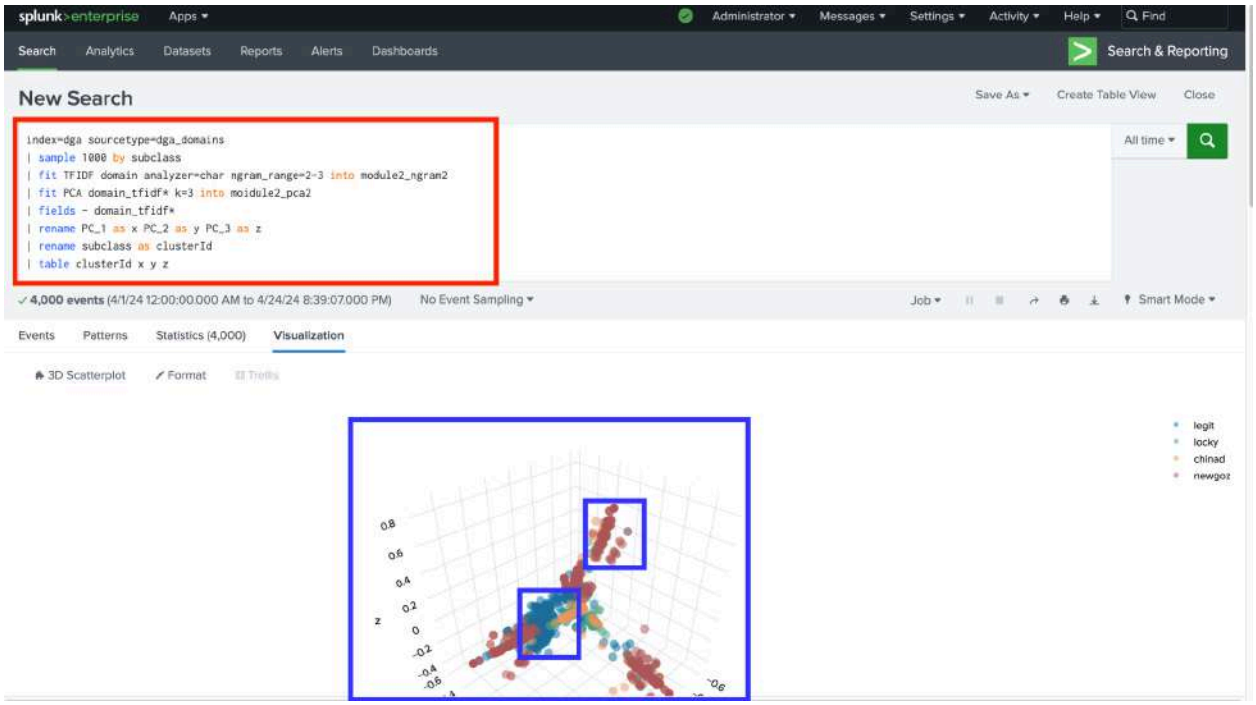

splunk>enterprise
Apps
Administrator
Messages
Settings
Activity
Help
Find
Search
Analytics
Datasets
Reports
Alerts
Dashboards
Search & Reporting
New Search
Save As
Create Table View
Close
index=dga sourcetype=dga_domains
| sample 1000 by subclass
| fit TFIDF domain analyzer=char ngram_range=2-3 into module2_ngram
| fit PCA domain_tfidf* k=3 into module2_pca
| fields - domain_tfidf*
| rename PC_1 as x PC_2 as y PC_3 as z
| rename class as clusterId
| table clusterId x y z
All time
[0] 100000 events (out of 100000) were missing the subclass field and were sampled as though they all had the same value.
1,000 events (4/1/24 12:00:00.000 AM to 4/24/24 8:38:31.000 PM)
No Event Sampling
Job
Smart Mode
Events
Patterns
Statistics (1,000)
Visualization
20 Per Page
Format
Preview
Prev
1
2
3
4
5
6
7
8
Next
clusterId
x
y
z
legit 0.007306490150871911 0.0842476228874978 0.1761781415291743
legit -0.20499848733229145 -0.1873736519535062 -0.08169904947705048
legit 0.37605229982279953 -0.13103349862388905 -0.03268474173438068
legit -0.14729138629387373 -0.1675341087303324 -0.06681534050635989
legit -0.3471448004272036 -0.17229574955950094 -0.07473872009932155
legit 0.33714046109334256 -0.034250917069783306 0.022029138234463317
legit -0.17502043952231888 -0.08472263200757857 -0.012402576941122153
legit -0.1874379475502906 -0.072662556416712 5 0.005536346055146083


splunk>enterprise
Apps
Administrator
Messages
Settings
Activity
Help
Find
Dashboards
Search & Reporting
Save As
Create Table View
Close
All time
dule2_ngram
field and were sampled as though they all had the same value.
M)
No Event Sampling
Job
Smart Mode
Splunk Visualizations
More
3D Scatterplot
Show relationships between discrete values in a three dimensional representation.
Search Fragment
| table clusterId x y z [clusterColor]
legit
dga
0.8
0.6
0.4
0.2
0
-0.2
z

5. Similarly, run the search below to conduct an N-gram analysis of the domains, but coloring based on subclass this time. Likewise, this search is reconstructing the `n-gram analysis (2-3 char groups) of domain names with PCA k=3 by subclass` panel. Again, visualize the data using the **3D Scatterplot** visualiation to make the data easier to interpret.

```
index=dga sourcetype=dga_domains
| sample 1000 by subclass
| fit TFIDF domain analyzer=char ngram_range=2-3 into module2_ngram2
| fit PCA domain_tfidf* k=3 into module2_pca
| fields - domain_tfidf*
| rename PC_1 as x PC_2 as y PC_3 as z
| rename subclass as clusterId
| table clusterId x y z
```

Note how even when stratifying and coloring by subclass there are clear groupings. These groupings might be easier to see after rotating the graph.



Since it's clear there are groupings or clusters of domains, we should be able to cluster the data based on additional features.

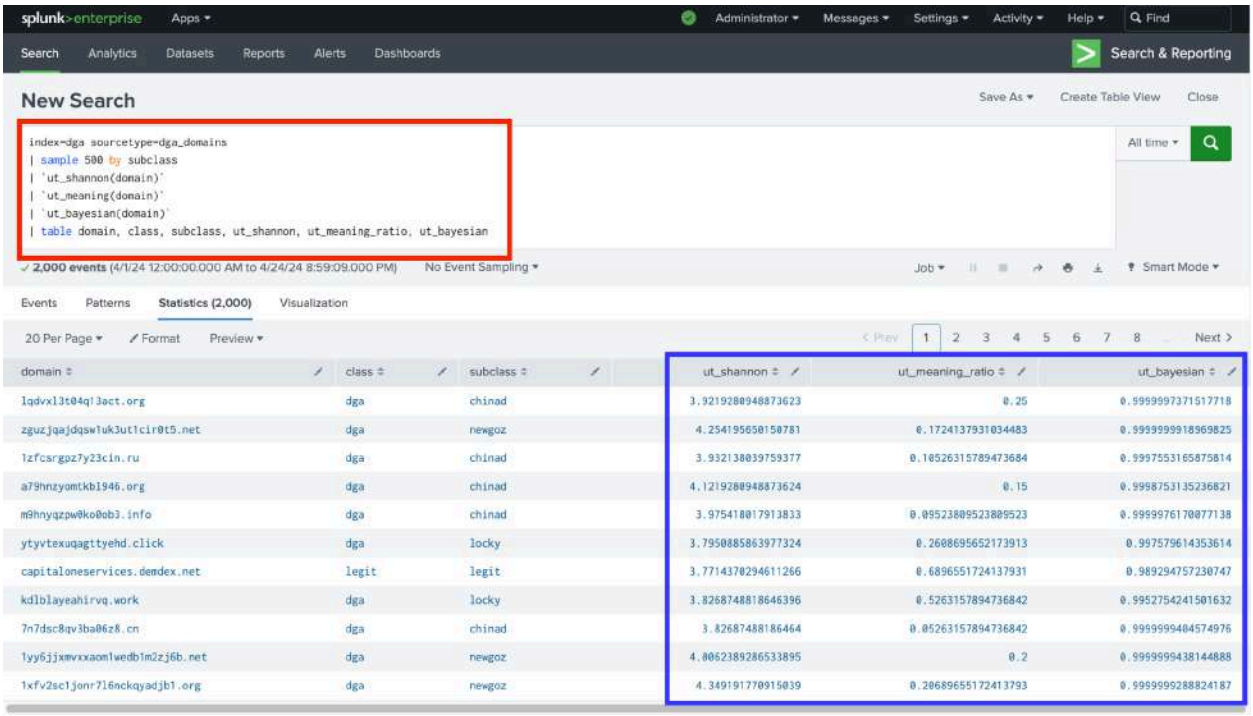
Task 2: Feature Engineering

In this module you will create new features that ML models can train from. Specifically, you'll be making extensive use of URL Toolbox, an app with custom SPL commands that can analyze domains and URLs. The goal of feature engineering here is to translate the domain into numeric features that can be used in training ML models.

1. In the browser tab open to the Botnet App for Splunk from Module 1, navigate to the **DGA App for Splunk**, then expand the **Dashboards** dropdown and select the **2. Feature Engineering and Selection** dashboard.
2. Run the following search to use URL Toolbox to add a shannon entropy, meaning, and Bayesian analysis score to the dataset:

```
index=dga sourcetype=dga_domains
| sample 500 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| table domain, class, subclass, ut_shannon, ut_meaning_ratio, ut_bayesian
```

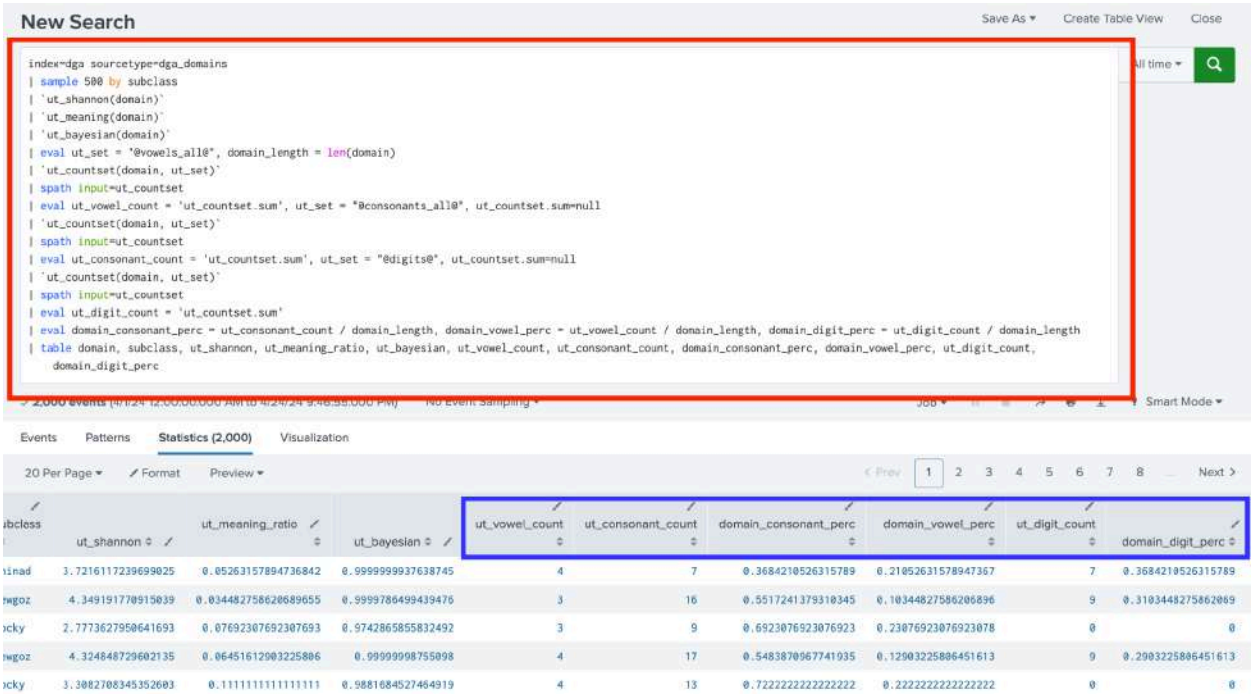
Note how a `ut_shannon`, `ut_meaning_ratio`,and `ut_bayesian` scores have been added to each event.



3. Run the following search to also add digit, consonant, and vowel counts and percentage values to the dataset.

```
index=dga sourcetype=dga_domains
| sample 500 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| table domain, subclass, ut_shannon, ut_meaning_ratio, ut_bayesian, ut_vowel_count,
ut_consonant_count, domain_consonant_perc, domain_vowel_perc, ut_digit_count,
domain_digit_perc
```

*Note how fields related to statistics around numeric, vowel, and consonant in the domain have been added to each event. You may need to scroll to the right in the **Statistics** table to view these new fields.*

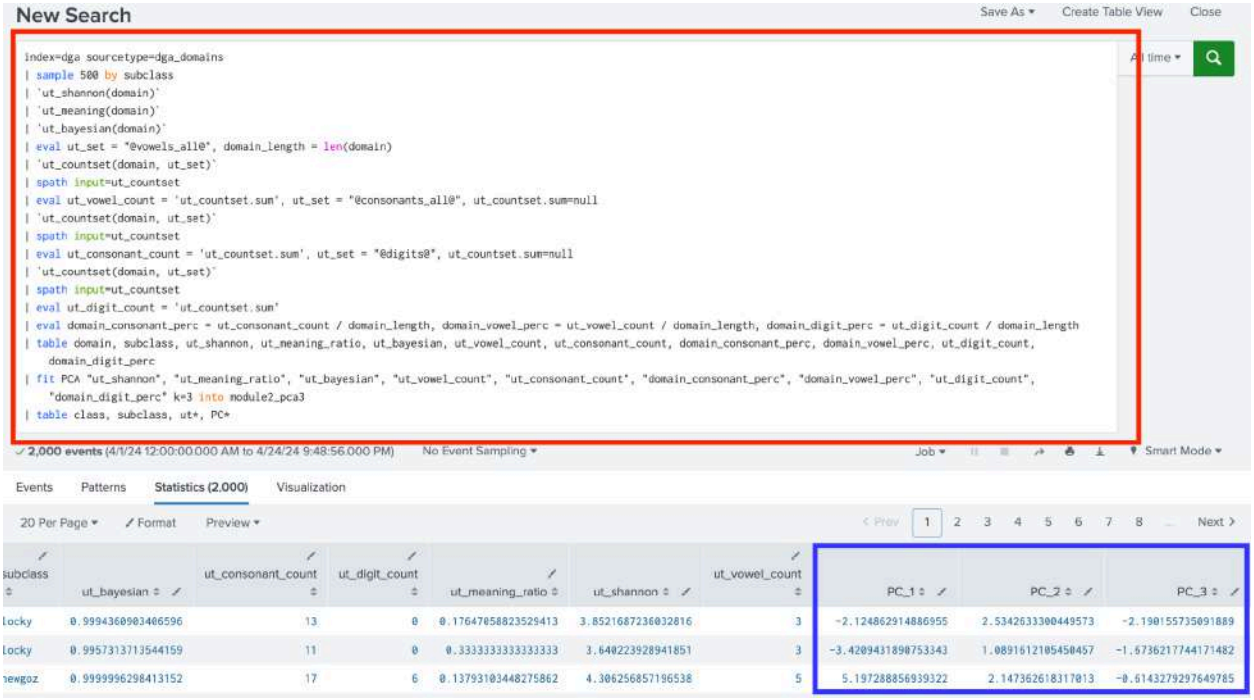


4. Run the following search to reduce the dimensions on the dataset using Principal Component Analysis (PCA), down from 9 dimensions down to 3. This search replicates the `Domain dataset enriched with features panel` on the `2. Feature Engineering and Selection` dashboard in the `DGA App for Splunk`.

```
index=dga sourcetype=dga_domains
| sample 500 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| table domain, subclass, ut_shannon, ut_meaning_ratio, ut_bayesian, ut_vowel_count,
ut_consonant_count, domain_consonant_perc, domain_vowel_perc, ut_digit_count,
domain_digit_perc
```

```
| fit PCA "ut_shannon", "ut_meaning_ratio", "ut_bayesian", "ut_vowel_count",
"ut_consonant_count", "domain_consonant_perc", "domain_vowel_perc", "ut_digit_count",
"domain_digit_perc" k=3 into module2_pca3
| table subclass, ut*, PC*
```

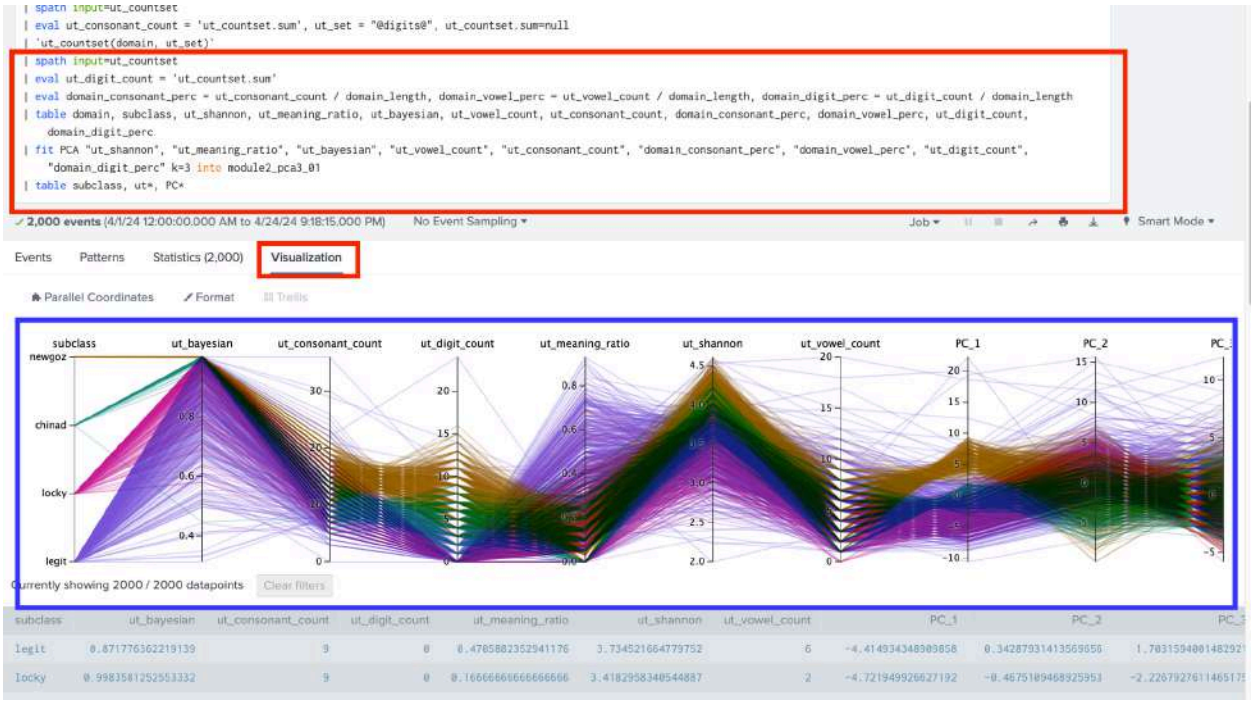
Note how three new fields have been added: `PC_1`, `PC_2`, `PC_3`. These are the reduced dimensions the PCA algorithm created.



5. Run the following search, then visualize the data with the **Parallel Coordinates** visualization. This visualization is closely replicating the `Parallel coordinate chart of classes and top features` visualization on the `2. Feature Engineering and Selection` dashboard in the `DGA App for Splunk`.

```
index=dga sourcetype=dga_domains
| sample 500 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
```

```
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| table domain, subclass, ut_shannon, ut_meaning_ratio, ut_bayesian, ut_vowel_count,
ut_consonant_count, domain_consonant_perc, domain_vowel_perc, ut_digit_count,
domain_digit_perc
| fit PCA "ut_shannon", "ut_meaning_ratio", "ut_bayesian", "ut_vowel_count",
"ut_consonant_count", "domain_consonant_perc", "domain_vowel_perc", "ut_digit_count",
"domain_digit_perc" k=3 into module2_pca3_01
| table subclass, ut*, PC*
```



Note how it's easy to see relationships between feature values using this visualization. For example, the domains generated by the `chinad` *and* `locky` *algorithms tend to have a high ut_bayesian score.*

Now that the additional features have been engineered, it's time to move onto the clustering component of this module.

Task 3: Clustering and Exploration

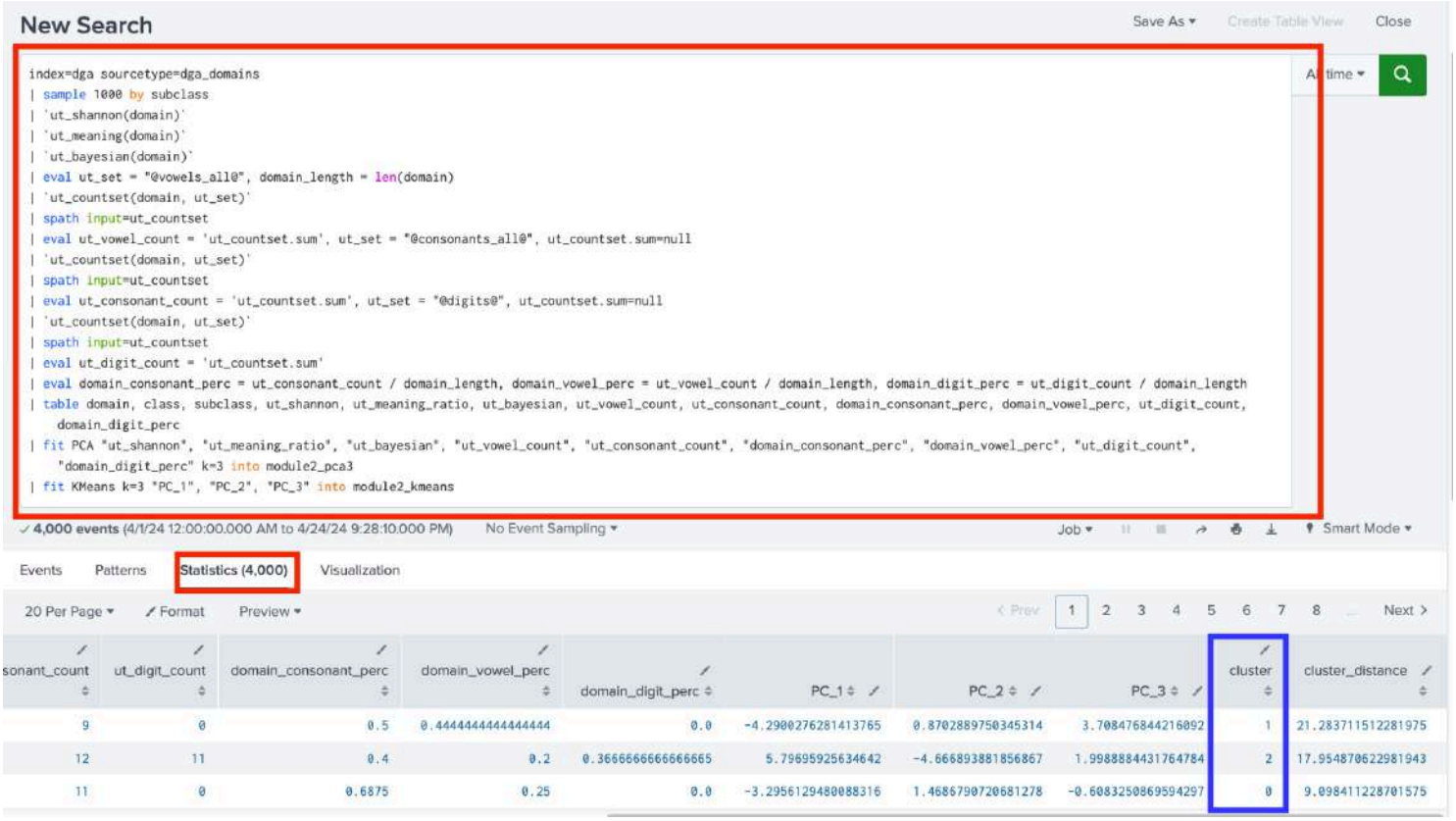
In this task, you will be using the features you engineered in Task 2 to cluster different domains together. There is no defined goal with clustering (unsupervised learning), but the idea is to see similarities between different classes by grouping them together in clusters.

1. Run the following search to cluster the data together, based on the PCA-reduced dimensions, and three possible clusters (as defined by the `K` hyperparameter).

```
index=dga sourcetype=dga_domains
| sample 1000 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| table domain, class, subclass, ut_shannon, ut_meaning_ratio, ut_bayesian,
ut_vowel_count, ut_consonant_count, domain_consonant_perc, domain_vowel_perc,
ut_digit_count, domain_digit_perc
| fit PCA "ut_shannon", "ut_meaning_ratio", "ut_bayesian", "ut_vowel_count",
"ut_consonant_count", "domain_consonant_perc", "domain_vowel_perc", "ut_digit_count",
"domain_digit_perc" k=3 into module2_pca3
| fit KMeans k=3 "PC_1", "PC_2", "PC_3" into module2_kmeans
```

*You might want to click back to the **Statistics** panel to see the numbers more easily.*

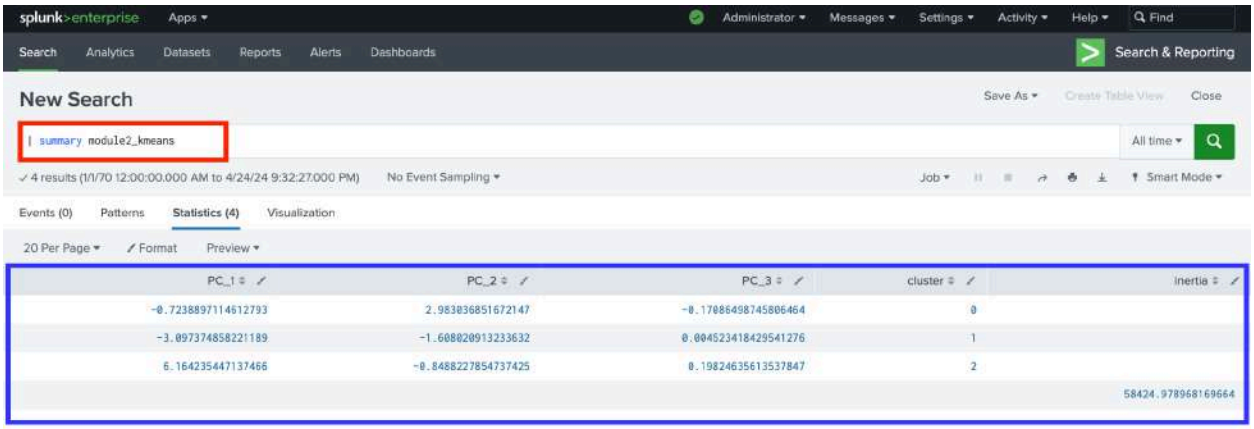
Note that the `cluster` field has been added to each event.



2. Run the following search to display the trained model parameters.

   ```
   | summary module2_kmeans
   ```

   Note that the parameters are all related to the different fields PCA created.



3. Run the search below, then view the results in a **3D Scatterplot** to view the clusters, colored by subclass.

```
index=dga sourcetype=dga_domains
| sample 1000 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| apply module2_pca3
| apply module2_kmeans
| rename cluster as clusterId
| rename PC_1 as x, PC_2 as y, PC_3 as z
| table clusterId x y z
```

Using the 3D Scatterplot visualization like this is a good way to view the clusters.

You may want to zoom in by scrolling and rotate the graph by clicking and dragging.

In this module you created new features based off of the DGA dataset, and in the next module you'll be training classification models to predict whether a given domain name is algorithmically-generated, or legitimate, as well as which algorithm generated the domain.

Module 3: Predictive Analysis

In this module, you will be using the same dataset and features you engineered from the previous module, but instead of clustering you will be training models to identify domain-generated algorithms. This can be a great way to identify hosts or users who are communicating with potentially malicious domains.

At first, you'll be training models to predict which algorithm generated the domain, but later on be training models to predict whether the domain is legitimate or algorithmically-generated. You will even be testing the prediction model against domains created by algorithms that aren't part of the training dataset!

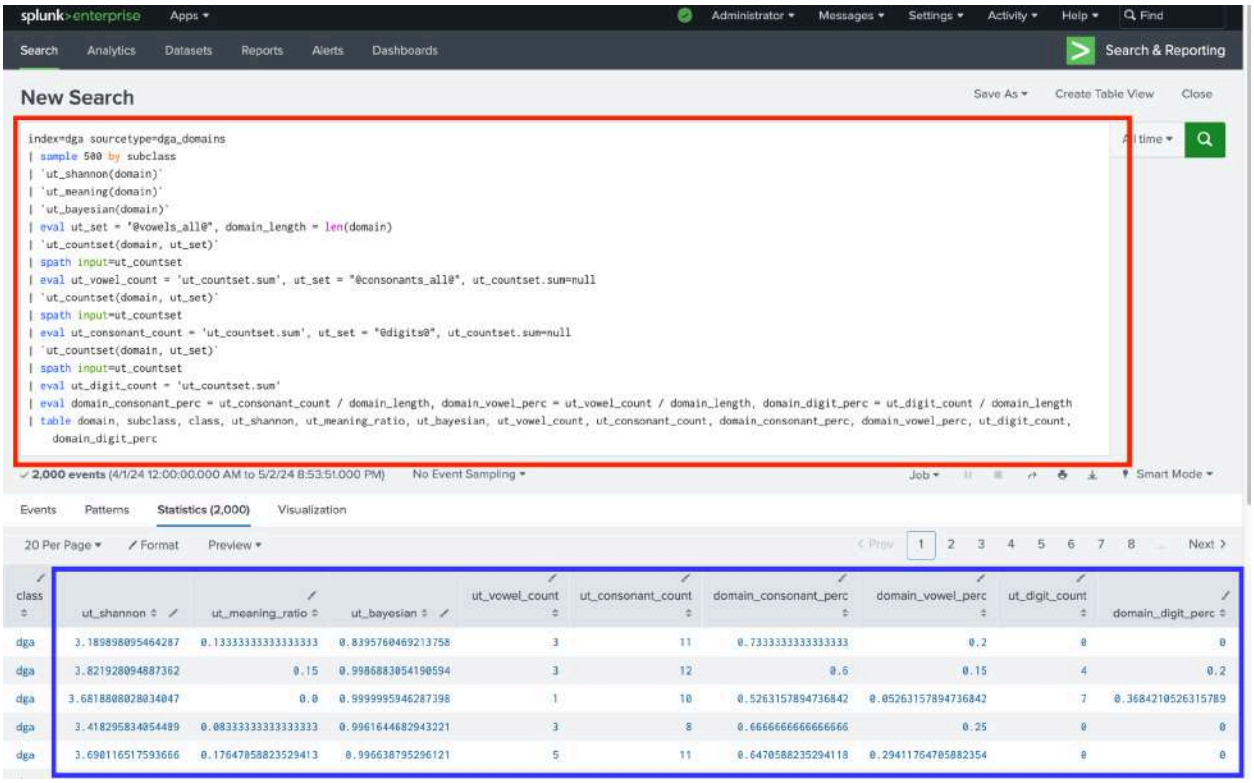
Task 1: Review Data and Engineered Features

In this task, you'll simply be reviewing the datasets and features engineered from Module 2.

1. In the browser tab open to the `Search & Reporting` app, run the following search to view the features that the models will be training against.

```
index=dga sourcetype=dga_domains
| sample 500 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| table domain, subclass, class, ut_shannon, ut_meaning_ratio, ut_bayesian,
ut_vowel_count, ut_consonant_count, domain_consonant_perc, domain_vowel_perc,
ut_digit_count, domain_digit_perc
```

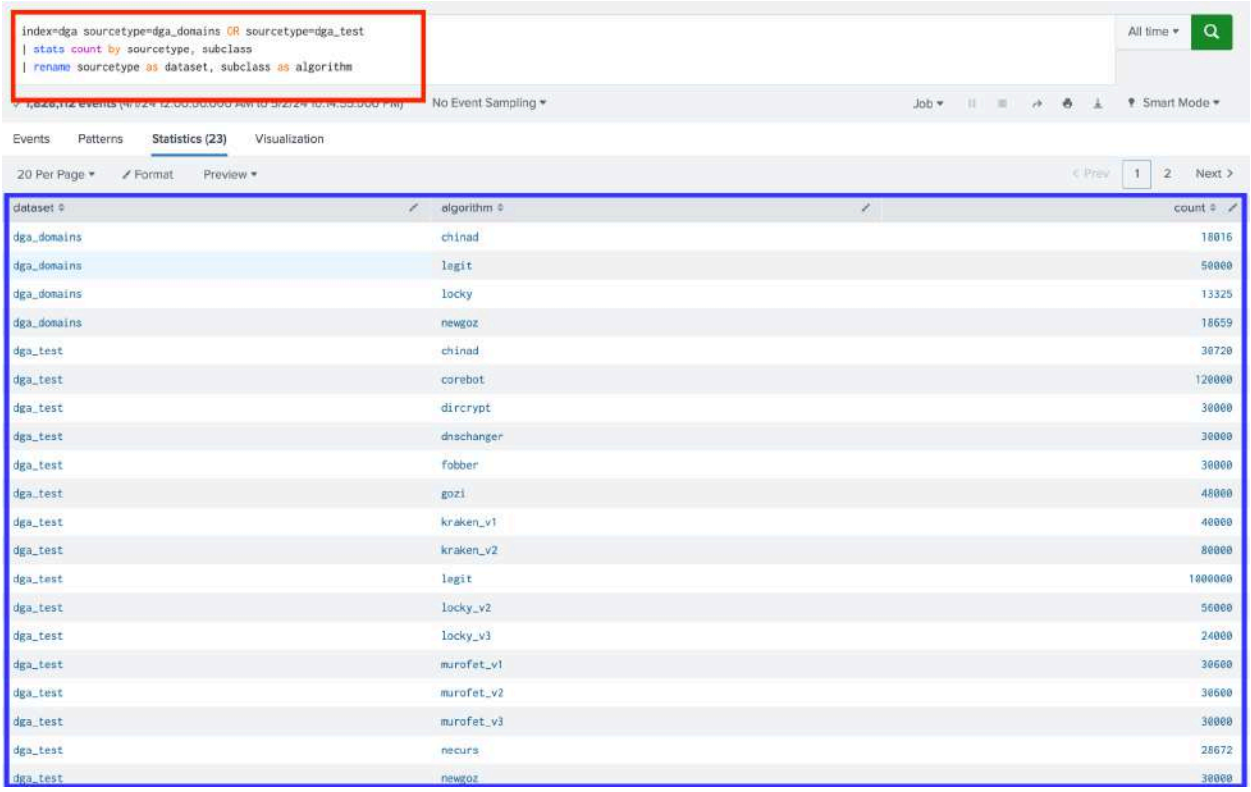
The `ut_shannon`, `ut_meaning_ratio`, `ut_bayesian`, `ut_vowel_count`, `ut_consonant_count`, `domain_consonant_perc`, `domain_vowel_perc`, `ut_digit_count`, `domain_digit_perc` *are the features the models will be training from.*



2. Run the search below to identify the counts of domains generated in each of the datasets (training and test, set as sourcetype).

```
index=dga sourcetype=dga_domains OR sourcetype=dga_test
| stats count by sourcetype, subclass
| rename sourcetype as dataset, subclass as algorithm
```

Note how there are more algorithms listed in the testing dataset than the training dataset. This will be important in Task 3.

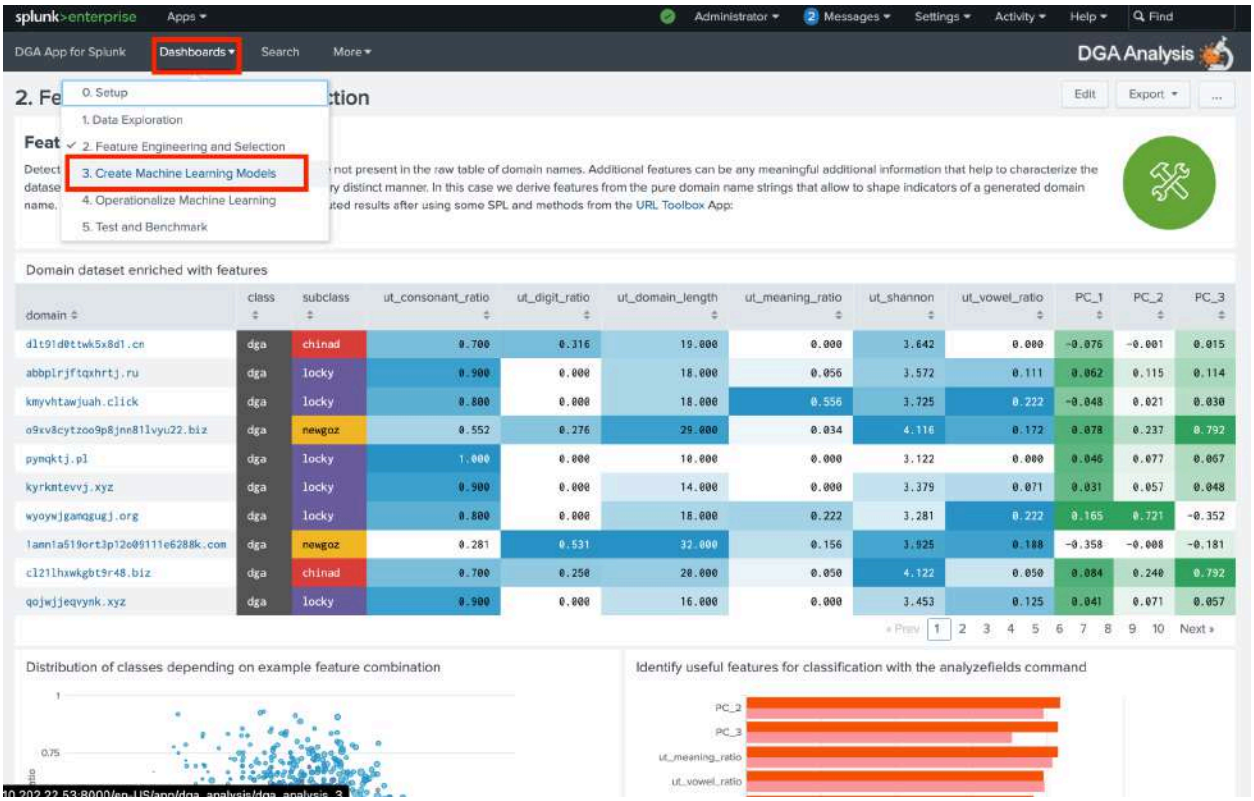


Task 2: Model Training, Testing, and Application for Algorithm Prediction

In this task, you will be training four different ML models against the training dataset to predict which algorithm generated the domain name (the `subclass` feature in the training and testing datasets). You will be training a random forest (RF), a support vector machine (SVM), a decision tree classifier (DCT), and a logistic regression (LR) model. It's often difficult to predict which machine learning algorithm will work best, which is why it can make sense to train multiple models at the same time then more forward with whichever performs the best.

1. In the browser tab open to the **DGA App for Splunk**, click **Dashboards** and select the **3. Create Machine Learning Models** dashboard. In the app, this is the dashboard that is used to train the models.

   If you receive an error about unsafe searches, feel free to ignore it.



2. In the browser tab open to the `Search & Reporting` app, run the following search to train all four models at once.

```
index=dga sourcetype=dga_domains
| sample 3000 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| fit RandomForestClassifier subclass from ut_vowel_count, ut_consonant_count,
ut_digit_count, ut_meaning_ratio, ut_shannon, ut_bayesian into module3_rf
| fit SVM subclass from ut_vowel_count, ut_consonant_count, ut_digit_count,
ut_meaning_ratio, ut_shannon, ut_bayesian into module3_svm
```

```
| fit DecisionTreeClassifier subclass from ut_vowel_count, ut_consonant_count,
ut_digit_count, ut_meaning_ratio, ut_shannon, ut_bayesian into module3_dtc
| fit LogisticRegression subclass from ut_vowel_count, ut_consonant_count,
ut_digit_count, ut_meaning_ratio, ut_shannon, ut_bayesian into module3_lr
```

In our testing, this search took about 75s to complete.

Feel free to ignore any search warnings such as those related to the 10,000 event limit.

Note how in each of the `fit` commands, the search lists the subclass field as the feature to predict.

Notice how all of the features that were engineered in Module 2 are being used to train the models.

Also, notice how this search is training four different models in the same search.

3. Run the following searches, each in turn to see the parameters for the models

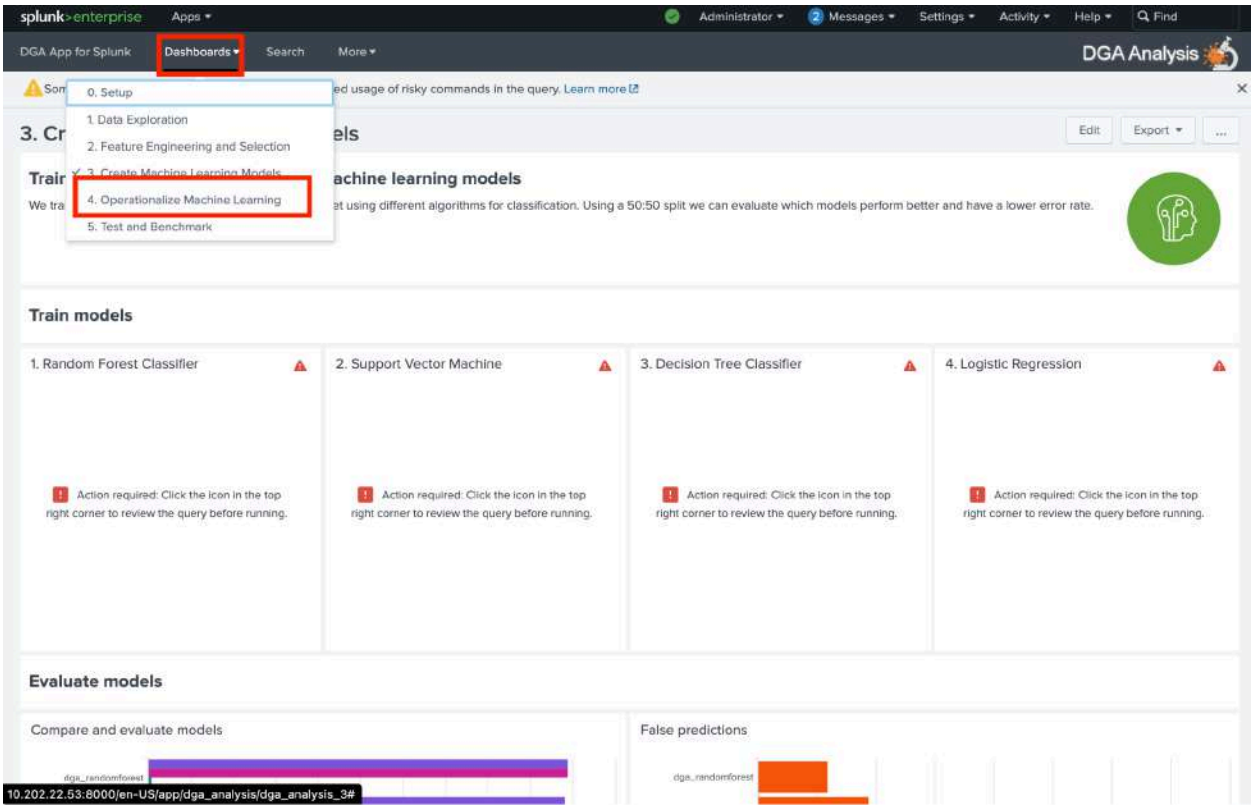
```
| summary module3_rf
```

```
| summary module3_dtc
```

```
| summary module3_lr
```

Note that the SVM model does not support summary, which is why it wasn't listed above as one to view the parameters for.

4. In the browser tab open to the **DGA App for Splunk**, click **Dashboards** and select the **4. Operationalize Machine Learning** dashboard. In the app, this is the dashboard that is used to test and operationalize the models predicting the algorithm generating the domain names.



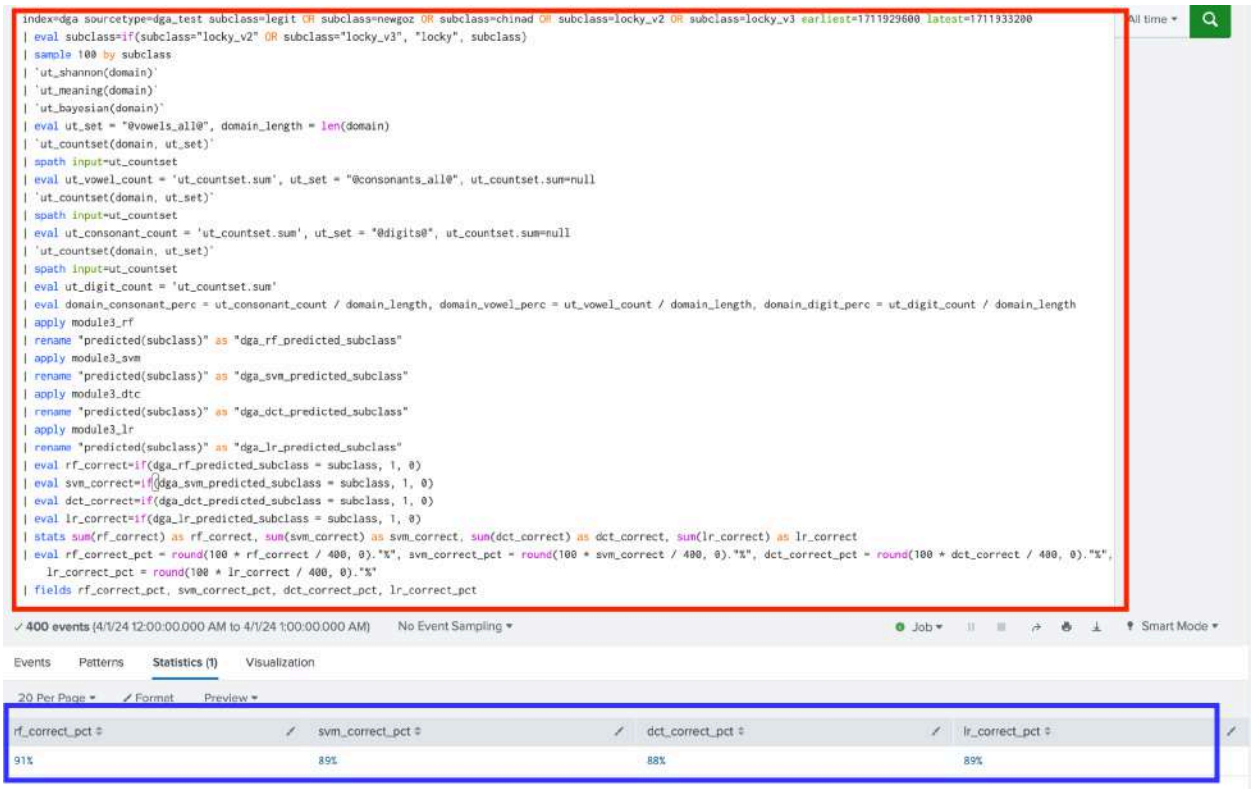
5. Run the following search to test the four models that were trained earlier in this task

```
index=dga sourcetype=dga_test subclass=legit OR subclass=newgoz OR subclass=chinad OR
subclass=locky_v2 OR subclass=locky_v3 earliest=1711929600 latest=1711933200
| eval subclass=if(subclass="locky_v2" OR subclass="locky_v3", "locky", subclass)
| sample 100 by subclass
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@", ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc =
ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| apply module3_rf
| rename "predicted(subclass)" as "dga_rf_predicted_subclass"
| apply module3_svm
| rename "predicted(subclass)" as "dga_svm_predicted_subclass"
```

```
| apply module3_dtc
| rename "predicted(subclass)" as "dga_dct_predicted_subclass"
| apply module3_lr
| rename "predicted(subclass)" as "dga_lr_predicted_subclass"
| eval rf_correct=if(dga_rf_predicted_subclass = subclass, 1, 0)
| eval svm_correct=if(dga_svm_predicted_subclass = subclass, 1, 0)
| eval dct_correct=if(dga_dct_predicted_subclass = subclass, 1, 0)
| eval lr_correct=if(dga_lr_predicted_subclass = subclass, 1, 0)
| stats sum(rf_correct) as rf_correct, sum(svm_correct) as svm_correct, sum(dct_correct) as
dct_correct, sum(lr_correct) as lr_correct
| eval rf_correct_pct = round(100 * rf_correct / 400, 0)."%", svm_correct_pct = round(100 *
svm_correct / 400, 0)."%", dct_correct_pct = round(100 * dct_correct / 400, 0)."%",
lr_correct_pct = round(100 * lr_correct / 400, 0)."%"
| fields rf_correct_pct, svm_correct_pct, dct_correct_pct, lr_correct_pct
```

Each column in the Statistics table shows the percentage of correct predictions each model made, when predicting which algorithm generated the domain (the `subclass` *feature).*

Not bad for a first attempt!



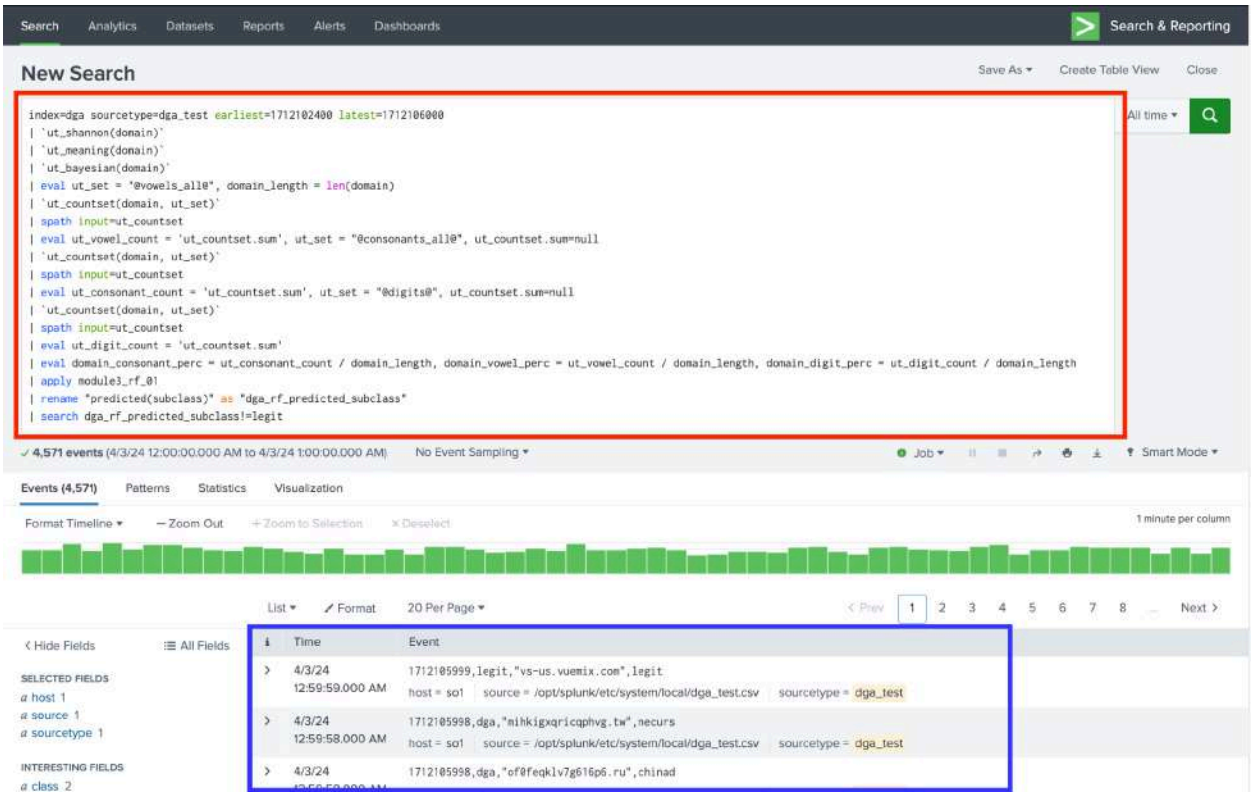
6. These models can be operationalized by using a search like the one below, to predict which algorithms generated a given domain or whether the domain is legitimate. This could even be created as an alert or ES correlation search to help identify devices communicating with domain-generated algorithms.

```
index=dga sourcetype=dga_test earliest=1712102400 latest=1712106000
| `ut_shannon(domain)`
| `ut_meaning(domain)`
```

```
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| apply module3_rf
| rename "predicted(subclass)" as "dga_rf_predicted_subclass"
| search dga_rf_predicted_subclass!=legit
```

In our testing, this search took about 45s to run.

This search is essentially finding any domain that isn't predicted to have a subclass of "legit", meaning that all results returned are predicted to be algorithmically-generated.



What if you wanted to predict just whether a domain was algorithmically-generated, and not specifically which algorithm generated the domain?

Task 3: Model Training, Testing, and Application for Class Prediction

Similar to the previous task, you'll be training, testing, and applying four different models in a set of searches. However, this time you will be training models only to predict whether the domain is legitimate or algorithmically-generated (the `class` feature in the testing and training datasets), not which algorithm generated the domain.

You will even be testing these models against algorithms that the models haven't been trained against, just to see how accurate they are. This is useful because in the real world, algorithms generating new domains may appear at any time and you won't have labeled datasets to train from.

1. Run the following search to train models to predict whether a given domain is legitimate (`legit`) or domain generated (`dga`).

```
index=dga sourcetype=dga_test earliest=1712102400 latest=1712106000
| sample 5000 by class
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| fit RandomForestClassifier class from ut_vowel_count, ut_consonant_count,
ut_digit_count, ut_meaning_ratio, ut_shannon, ut_bayesian into module3_rf2
| fit SVM class from ut_vowel_count, ut_consonant_count, ut_digit_count,
ut_meaning_ratio, ut_shannon, ut_bayesian into module3_svm2
| fit DecisionTreeClassifier class from ut_vowel_count, ut_consonant_count,
ut_digit_count, ut_meaning_ratio, ut_shannon, ut_bayesian into module3_dtc2
| fit LogisticRegression class from ut_vowel_count, ut_consonant_count,
ut_digit_count, ut_meaning_ratio, ut_shannon, ut_bayesian into module3_lr2
```

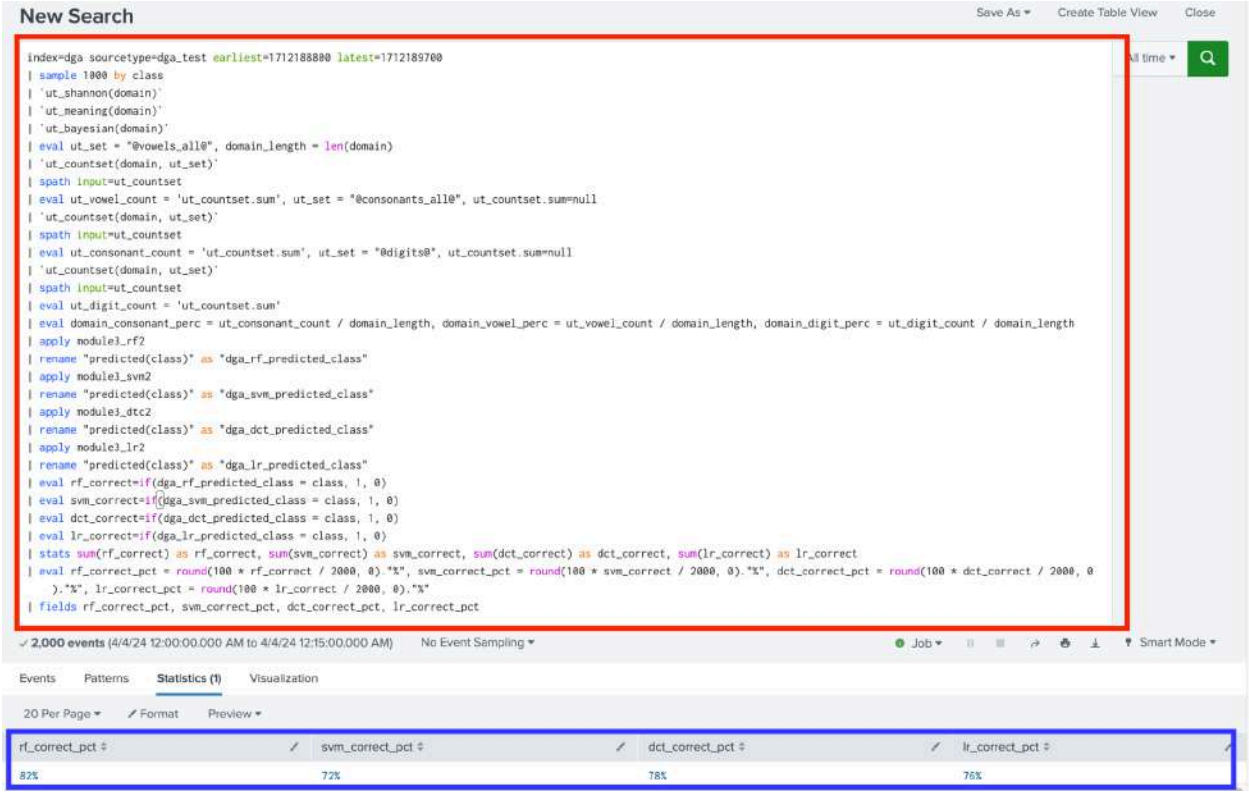
In our testing, this search took about 65s to run.

Note how in each of the `fit` commands, the search lists the `class` field as the one to predict.

2. In the browser tab open to the **DGA App for Splunk**, click **Dashboards** and select the **5. Test and Benchmark** dashboard. In the app, this is the dashboard that is used to test the models against many algorithms including many not in the testing dataset.
3. Run the following search to measure how well the four models do against all of the algorithms in the testing dataset

```
index=dga sourcetype=dga_test earliest=1712188800 latest=1712189700
| sample 1000 by class
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| apply module3_rf2
| rename "predicted(class)" as "dga_rf_predicted_class"
| apply module3_svm2
| rename "predicted(class)" as "dga_svm_predicted_class"
| apply module3_dtc2
| rename "predicted(class)" as "dga_dct_predicted_class"
| apply module3_lr2
| rename "predicted(class)" as "dga_lr_predicted_class"
| eval rf_correct=if(dga_rf_predicted_class = class, 1, 0)
| eval svm_correct=if(dga_svm_predicted_class = class, 1, 0)
| eval dct_correct=if(dga_dct_predicted_class = class, 1, 0)
| eval lr_correct=if(dga_lr_predicted_class = class, 1, 0)
| stats sum(rf_correct) as rf_correct, sum(svm_correct) as svm_correct,
sum(dct_correct) as dct_correct, sum(lr_correct) as lr_correct
| eval rf_correct_pct = round(100 * rf_correct / 2000, 0)."%", svm_correct_pct =
round(100 * svm_correct / 2000, 0)."%", dct_correct_pct = round(100 * dct_correct /
2000, 0)."%", lr_correct_pct = round(100 * lr_correct / 2000, 0)."%"
| fields rf_correct_pct, svm_correct_pct, dct_correct_pct, lr_correct_pct
```

Notice how the correctly guessed percentages have dropped from about 90% to about 80%. This is likely because there are domains in this training dataset the algorithms have never seen before.



4. Run the search below to create a confusion matrix for just the random forest model

```
index=dga sourcetype=dga_test earliest=1712188800 latest=1712189700
| sample 1000 by class
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| apply module3_rf2
| rename "predicted(class)" as "dga_rf_predicted_class"
| apply module3_svm2
| rename "predicted(class)" as "dga_svm_predicted_class"
| apply module3_dtc2
```

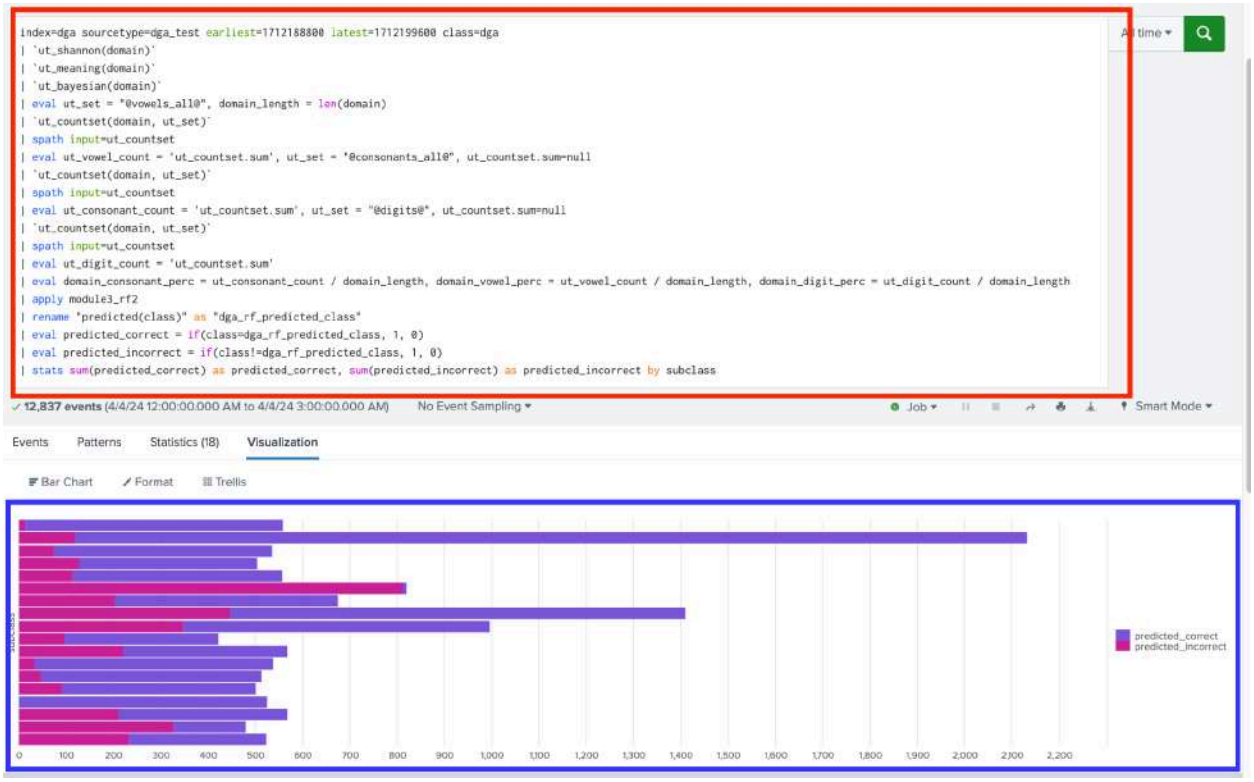
```
| rename "predicted(class)" as "dga_dct_predicted_class"
| apply module3_lr2
| rename "predicted(class)" as "dga_lr_predicted_class"
| eval tp = if(class="dga" AND dga_rf_predicted_class="dga", 1, 0)
| eval fp = if(class="legit" AND dga_rf_predicted_class="dga", 1, 0)
| eval tn = if(class="legit" AND dga_rf_predicted_class="legit", 1, 0)
| eval fn = if(class="dga" AND dga_rf_predicted_class="legit", 1, 0)
| stats sum(tp) as true_positives, sum(fp) as false_positives, sum(tn) as
true_negatives, sum(fn) as false_negatives
```

5. Run the search below to see the number of correct and incorrect predictions made by the random forest model stratified by which algorithm generated the domain (`subclass` feature) in the testing dataset. Use the **Bar Chart** visualization, and **Format** the visualization so that the bars are **stacked.** This is effectively re-creating the left panel in the second row from the top on the `5. Test and Benchmark` dashboard in the `DGA App for Splunk`.

```
index=dga sourcetype=dga_test earliest=1712188800 latest=1712189700 class=dga
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
```

```
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| apply module3_rf2
| rename "predicted(class)" as "dga_rf_predicted_class"
| eval predicted_correct = if(class=dga_rf_predicted_class, 1, 0)
| eval predicted_incorrect = if(class!=dga_rf_predicted_class, 1, 0)
| stats sum(predicted_correct) as predicted_correct, sum(predicted_incorrect) as
predicted_incorrect by subclass
```

Note how the model was not very successful in identifying domains generated by some algorithms it didn't train against, like the `gozi` *algorithm, but it was reasonably successful in predicting domains generated by other algorithms it has not been trained against like* `proslikefan`.

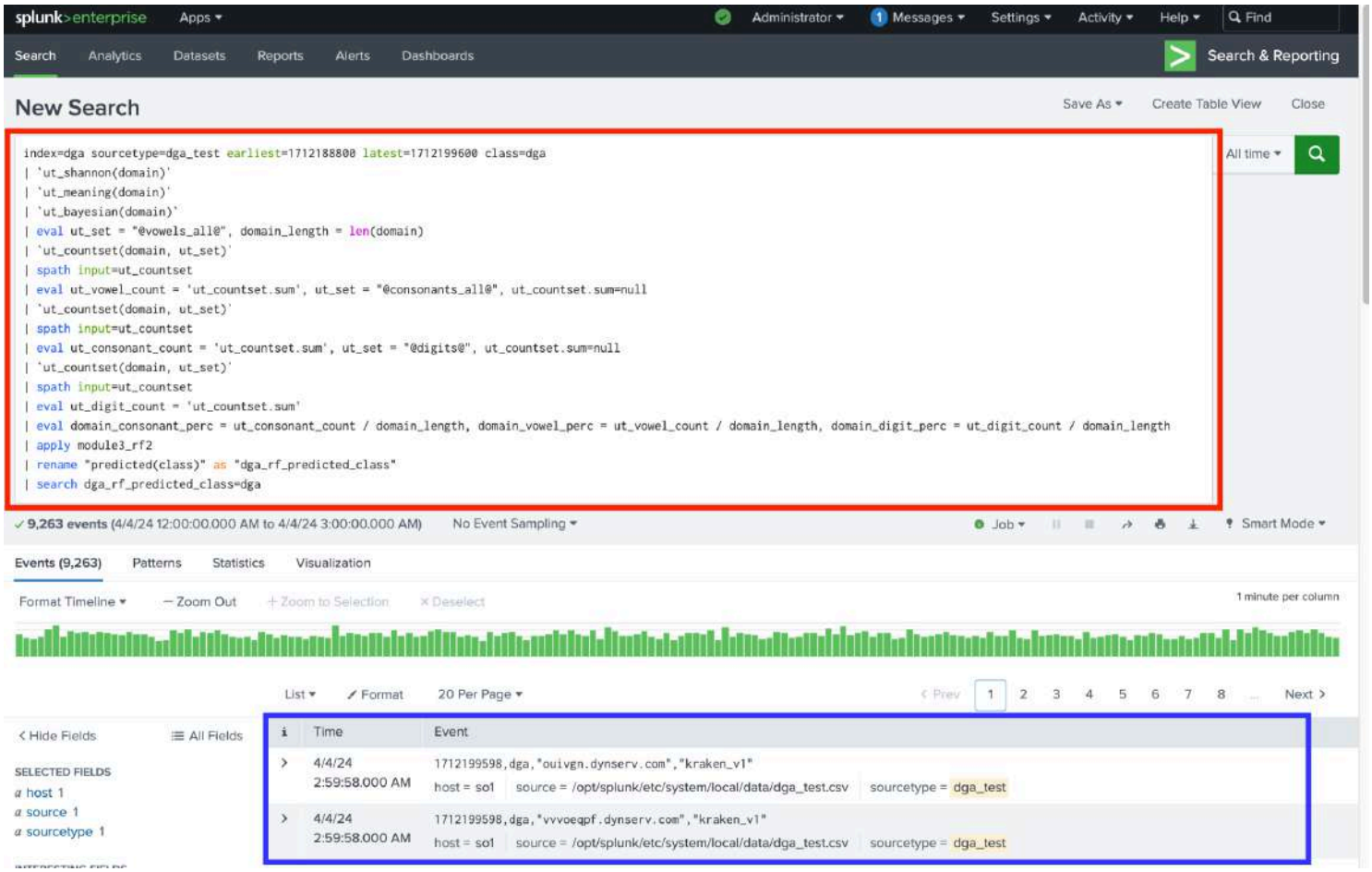


6. This model to only identify whether a domain is algorithmically-generated can be operationalized in a search like the one below. Just like with the previous task this search could be created as an alert or ES correlation search to help identify devices communicating with domain-generated algorithms.

```
index=dga sourcetype=dga_test earliest=1712188800 latest=1712199600 class=dga
| `ut_shannon(domain)`
| `ut_meaning(domain)`
| `ut_bayesian(domain)`
| eval ut_set = "@vowels_all@", domain_length = len(domain)
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
```

```
| eval ut_vowel_count = 'ut_countset.sum', ut_set = "@consonants_all@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_consonant_count = 'ut_countset.sum', ut_set = "@digits@",
ut_countset.sum=null
| `ut_countset(domain, ut_set)`
| spath input=ut_countset
| eval ut_digit_count = 'ut_countset.sum'
| eval domain_consonant_perc = ut_consonant_count / domain_length, domain_vowel_perc
= ut_vowel_count / domain_length, domain_digit_perc = ut_digit_count / domain_length
| apply module3_rf2
| rename "predicted(class)" as "dga_rf_predicted_class"
| search dga_rf_predicted_class=dga
```

This search is looking for any domains where the predicted `class` *is* `dga`*, meaning any domain generated by an algorithm.*



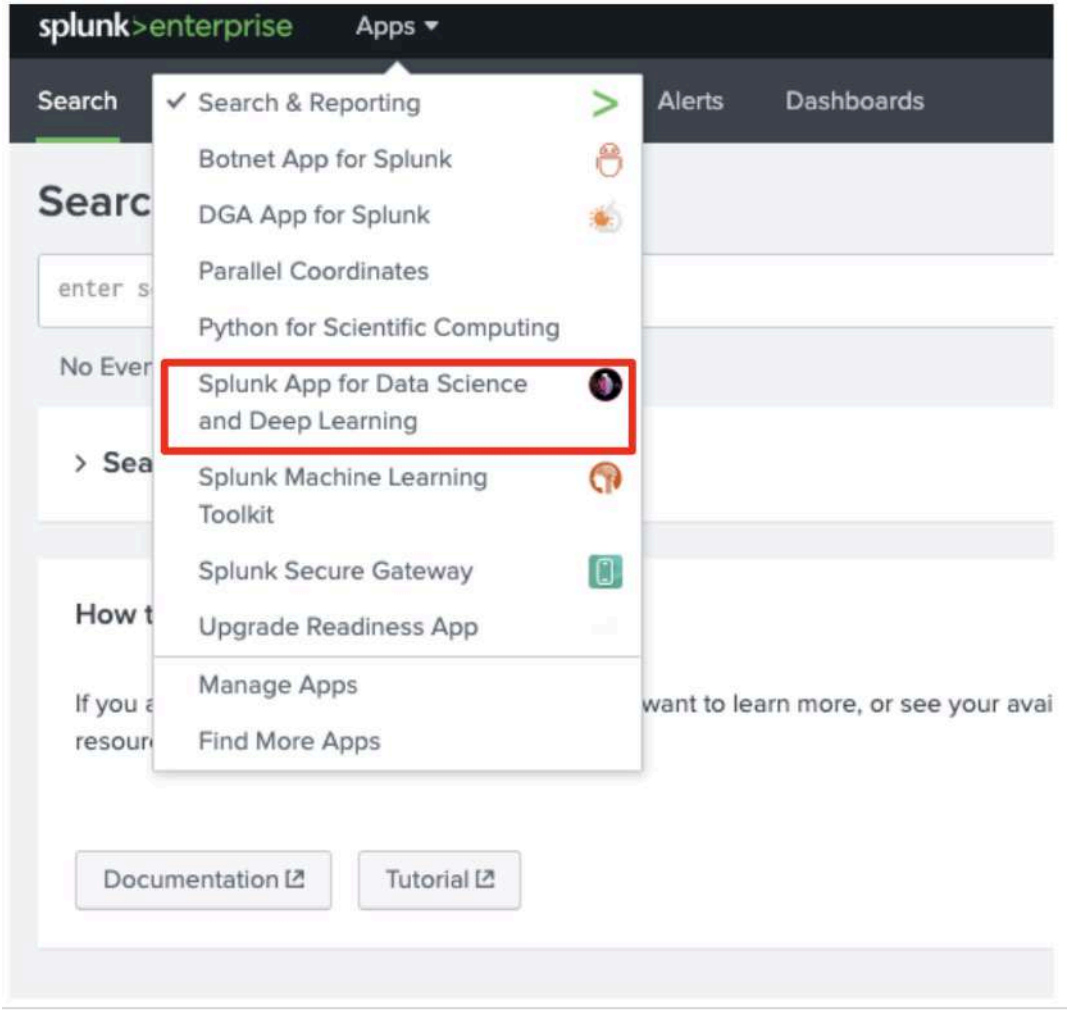
In this module you trained multiple models to predict not only which algorithm generated a domain, but also to predict whether a domain was generated by an algorithm. You also benchmarked and test against domains generated by algorithms the model had never seen before.

Module 4: Data Science & Deep Learning

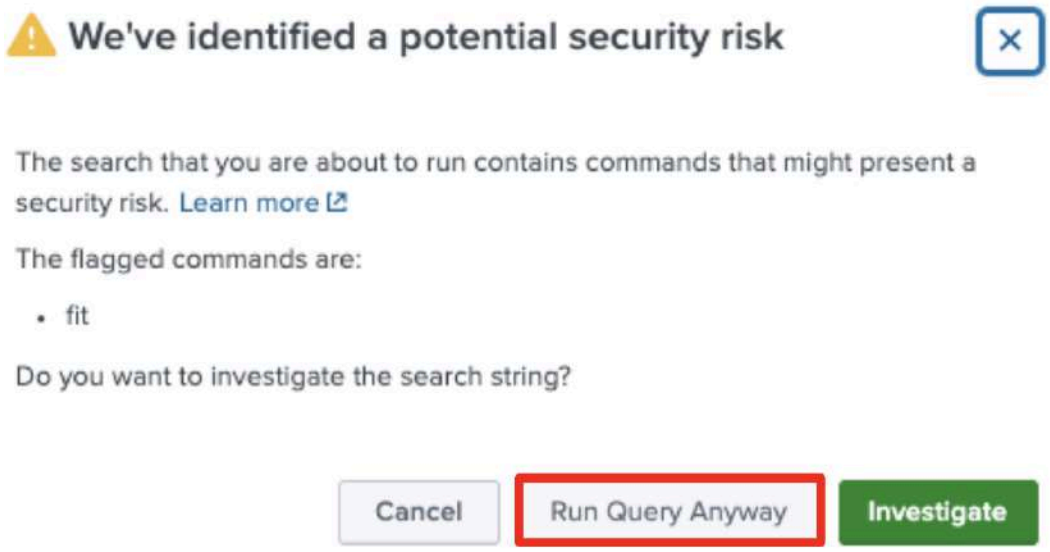
In this module you will further explore Splunks custom AI capabilities using the Data Science and Deep Learning (DSDL) app which enables deep learning modeling through a containerized deep learning environment and integration with Jupyter notebooks.

Task 1: Verify DSDL Setup

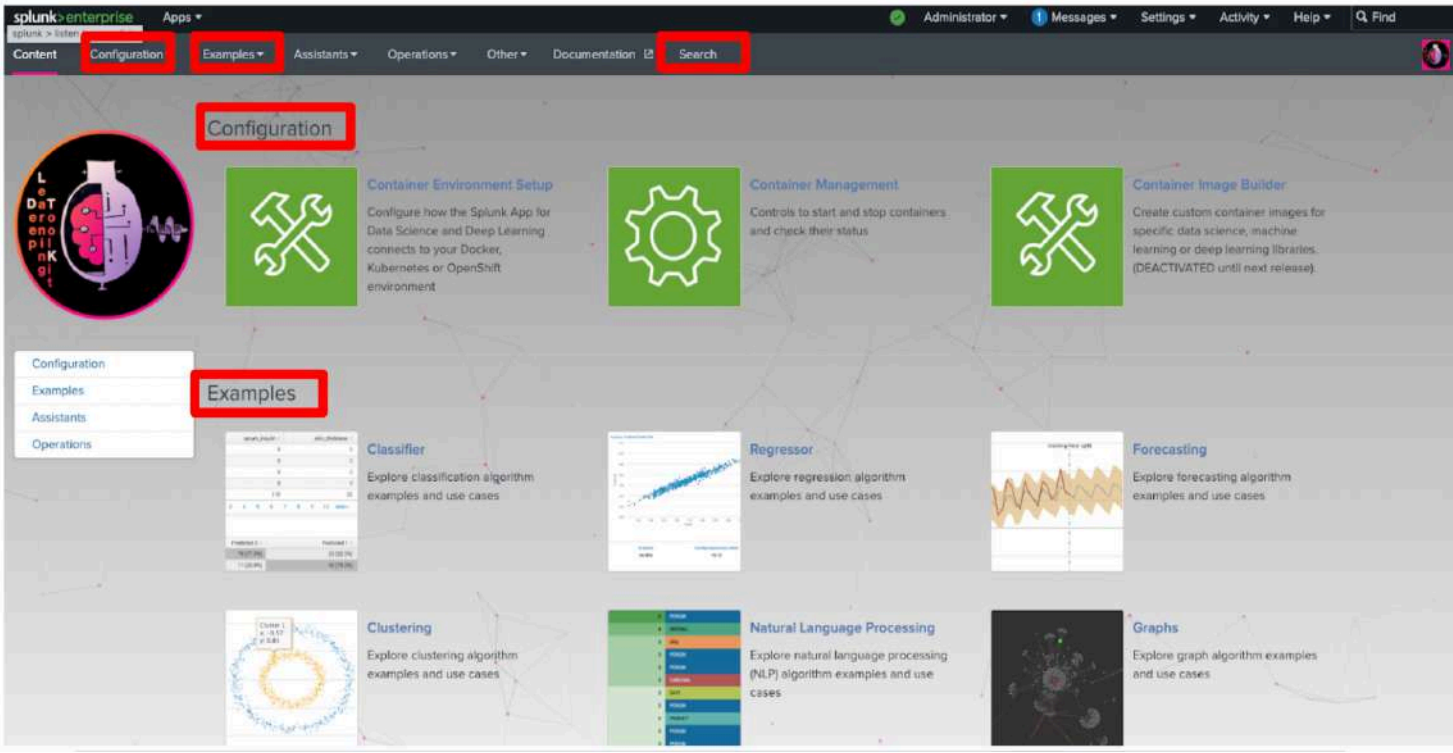
1. Navigate to the DSDL application



*NOTE: If you receive the message window below at anytime, please click the "**Run Query Anyway**" button*

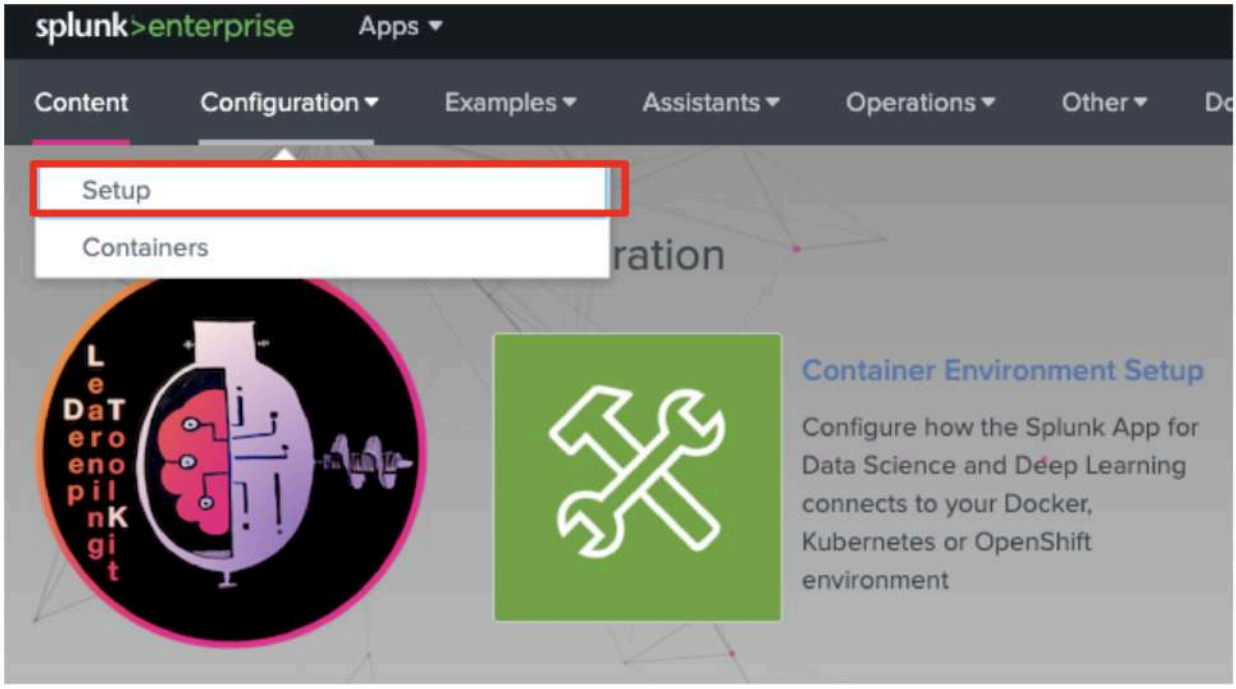


2. You should now be at the DSDL home page. Take a moment to familiarize yourself with the tool bar options, especially (**Configuration**, **Examples**, and **Search**). Don't forget to scroll down, to see the full menu of options.

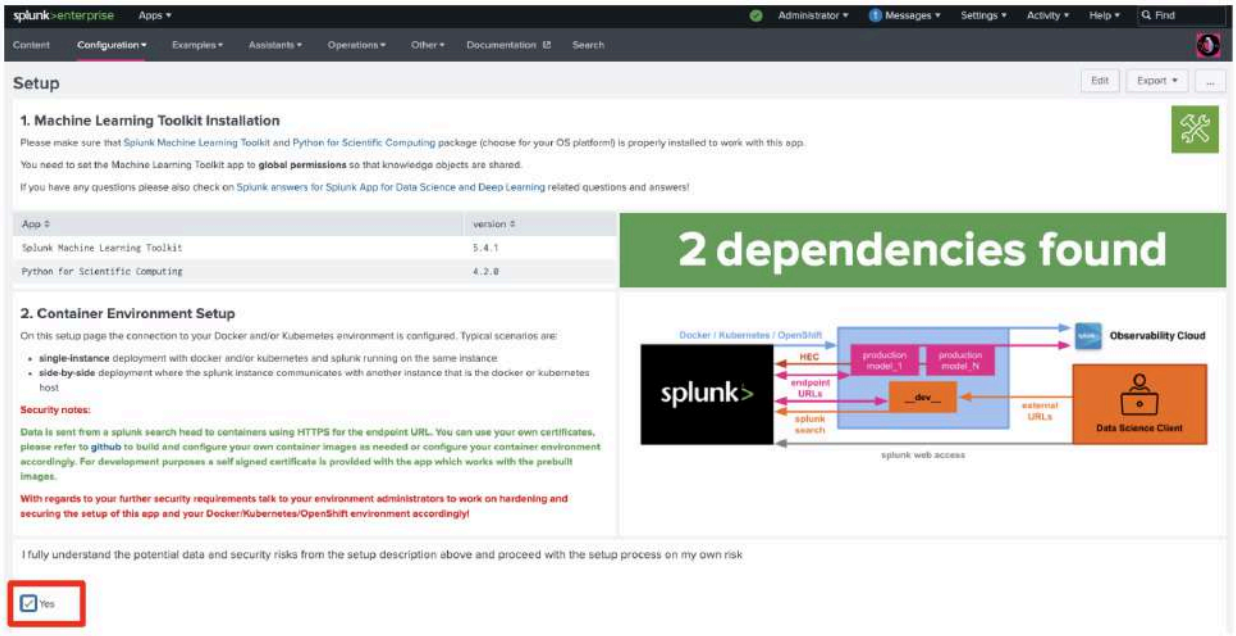


3.

4. Let's look at the container settings: **Configuration > Setup**

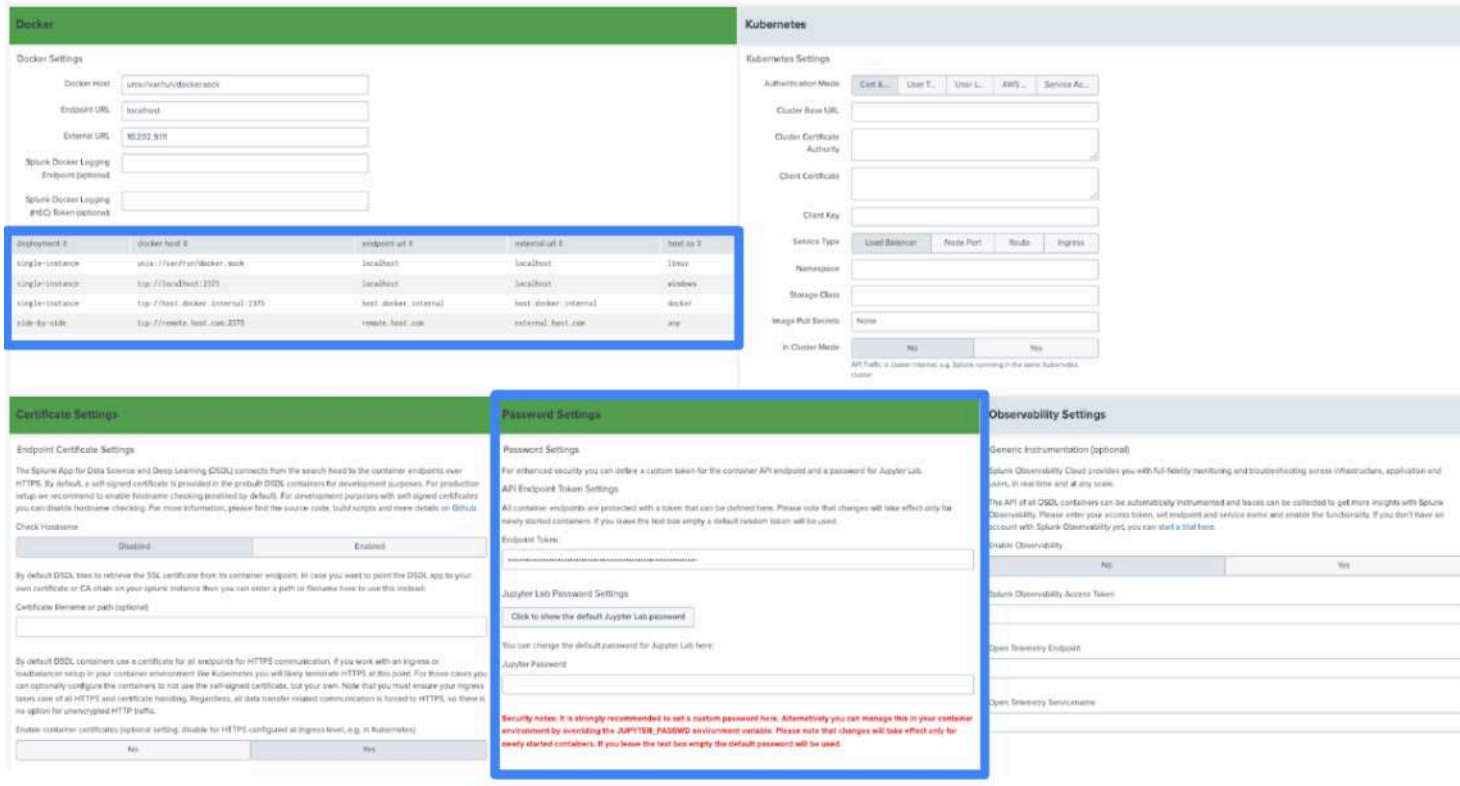


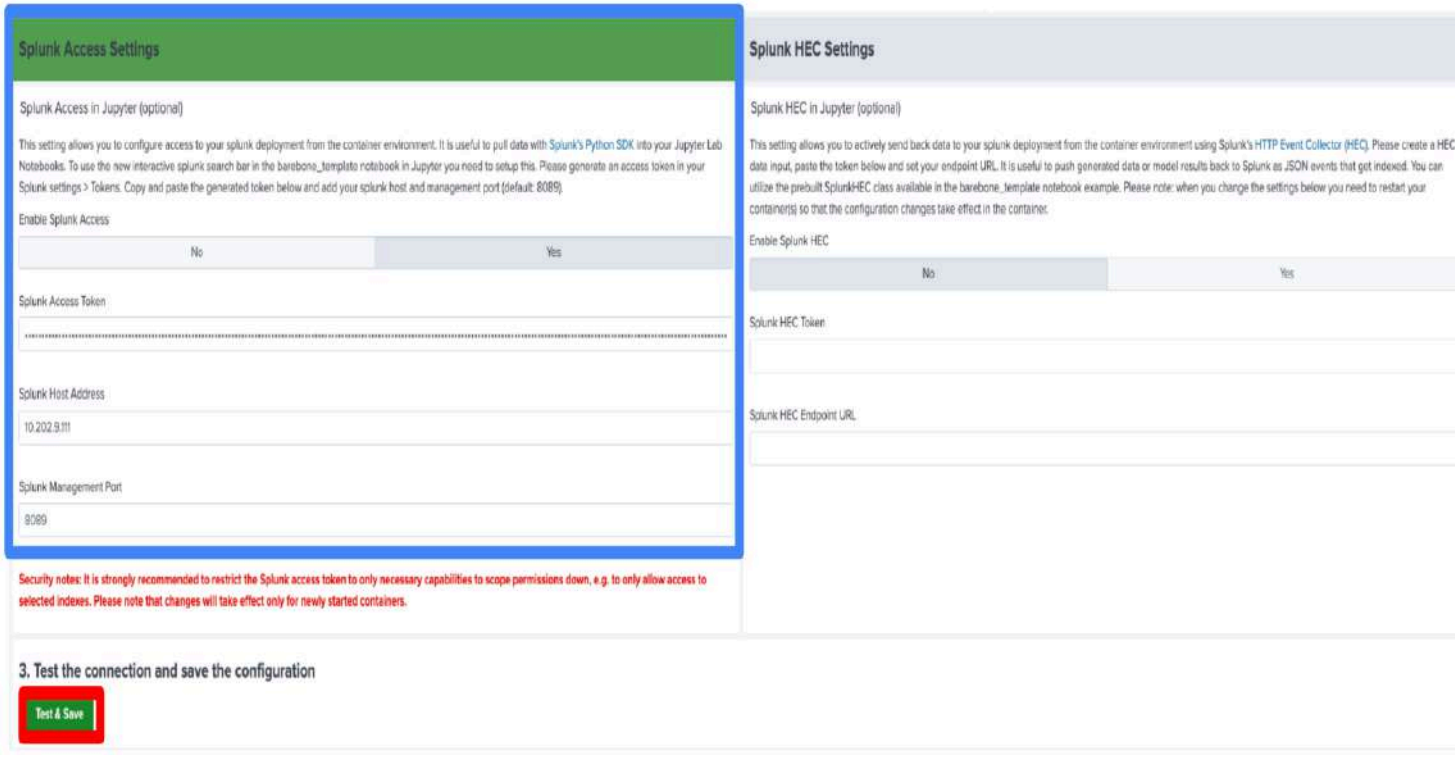
5. Now, take a closer look at the container setup. Click the "**yes**" box scroll down



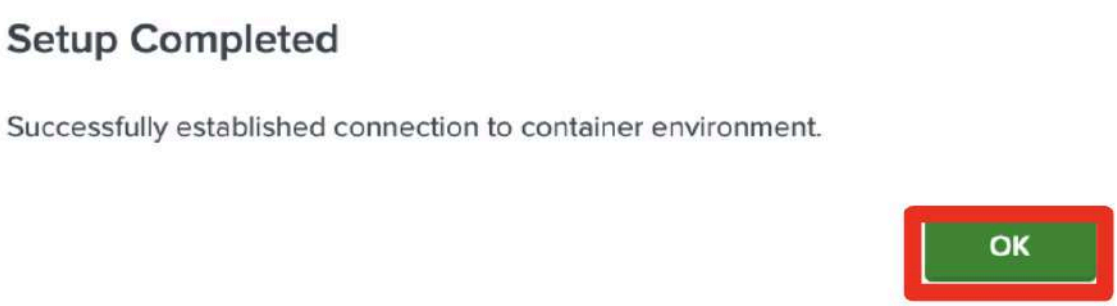
6. Verify the Docker container setup (*make sure to scroll down*) to ensure that the *Docke*r container information, *Password Settings*, and *Splunk Access Settings* are populated similar to the images below, and then click **Test & Save**.

   The DSDL enables you to utilize a containerized compute (CPU or GPU) environment for heavy modeling processing. In a production environment best practice would be to utilize Docker in a Kubernetes environment such as AWS EKS, but in this lab you will be using a Docker container running on your local instance.

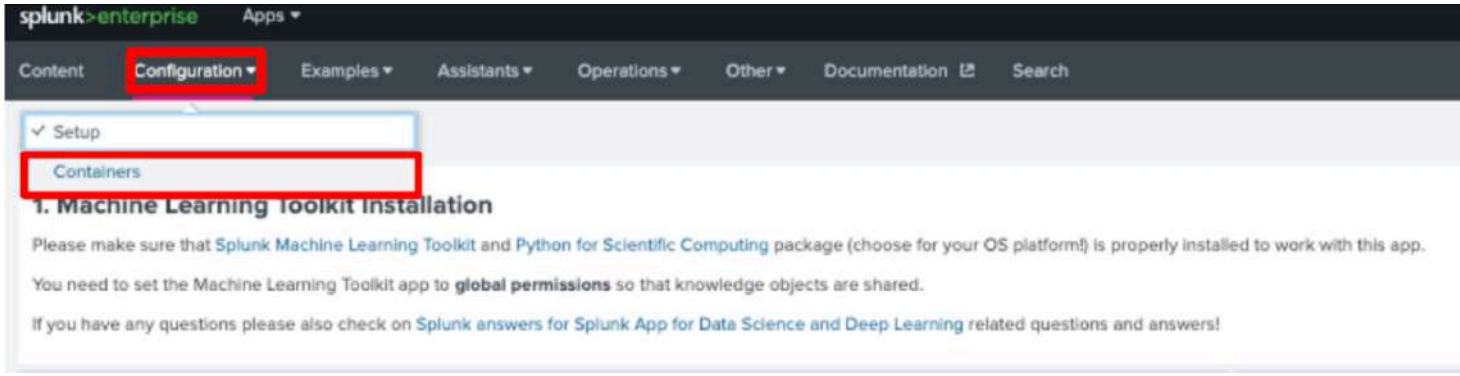

Docker
Docker Settings
Kubernetes
Kubernetes Settings
Certificate Settings
Endpoint Certificate Settings
Password Settings
Password Settings
Observability Settings


Splunk Access Settings
Splunk Access in Jupyter (optional)
Enable Splunk Access
No
Yes
Splunk Access Token
Splunk Host Address
10.202.9.111
Splunk Management Port
8089
Splunk HEC Settings
Splunk HEC in Jupyter (optional)
Enable Splunk HEC
No
Yes
Splunk HEC Token
Splunk HEC Endpoint URL
Security notes: It is strongly recommended to restrict the Splunk access token to only necessary capabilities to scope permissions down, e.g. to only allow access to selected indexes. Please note that changes will take effect only for newly started containers.
3. Test the connection and save the configuration
Test & Save

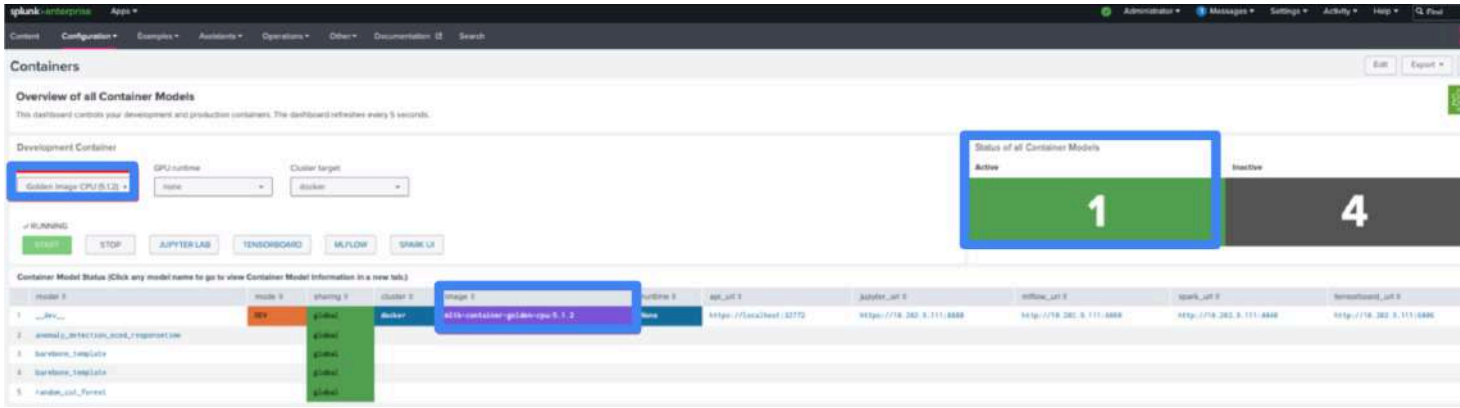
7. Wait for the *setup completed* window to appear, and then click **OK**



8. Confirm container is running. Click on **Configuration > Containers**



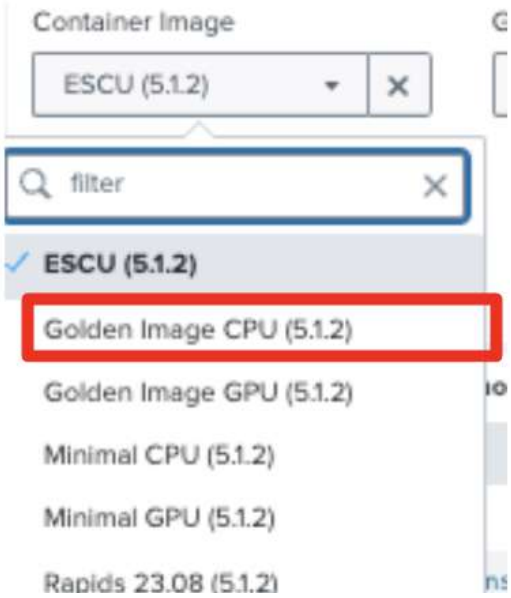
9. Verify you have the right container running “Golden Image CPU (5.1.2)”. You should see one container showing **Active**.



If you do not see the correct image Golden Image CPU (5.1.2), perform steps 9.a - 9.c. Else, skip to 8

a. Click stop

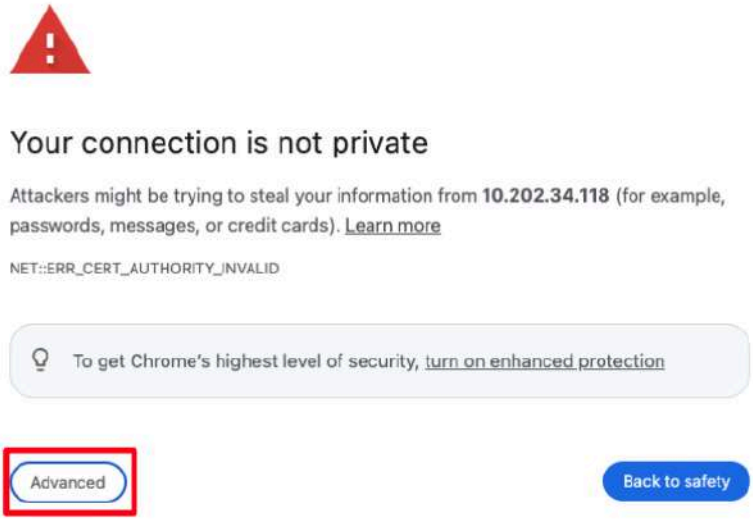
b. Select the Golden Image CPU (5.1.2)



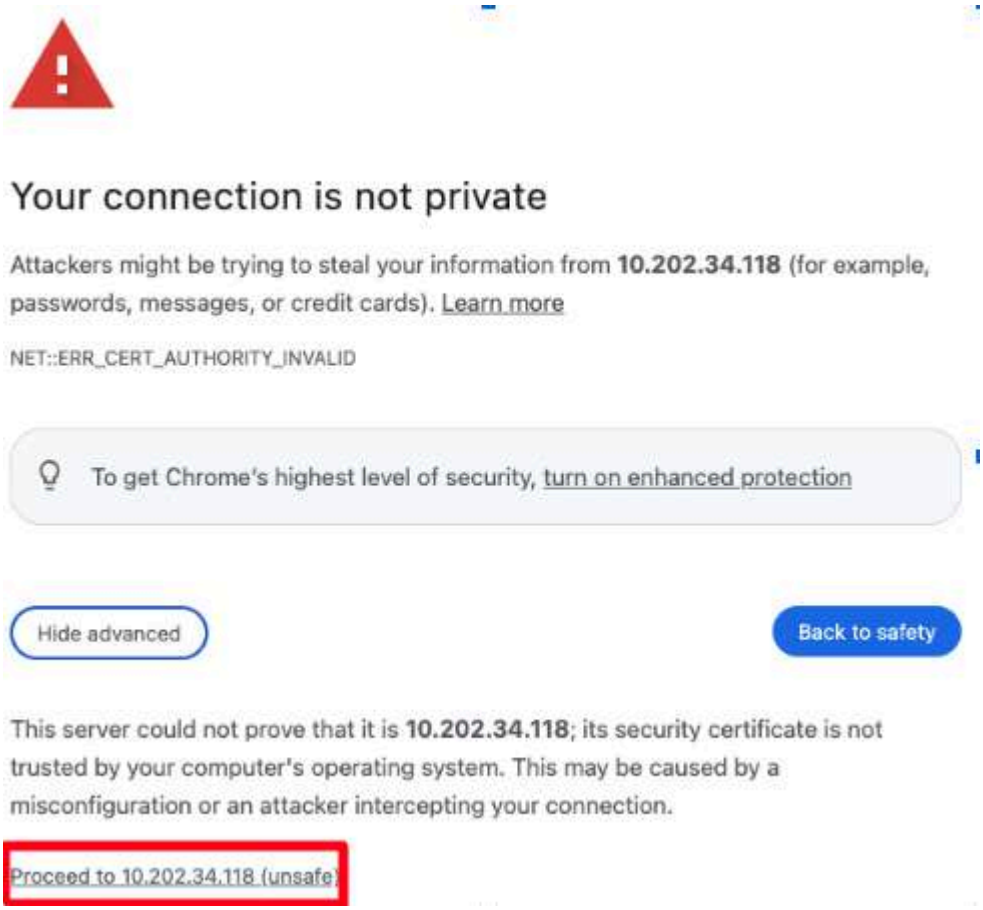
c. Click start

10. Click on **Jupyter Lab**

a. If you receive this window, click on **Advanced**



b. Then click the "**Proceed to…**" link



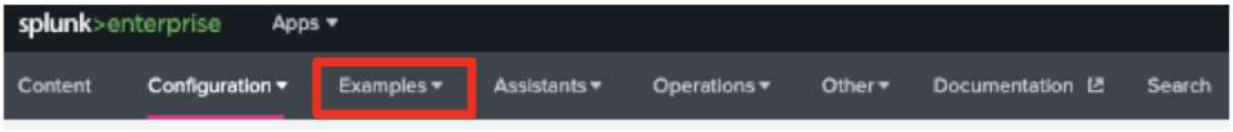
11. A new tab will open. Enter your `Password`: `Splunk4DeepLearning`

Task 2: Review the Example results in the DSDL interface

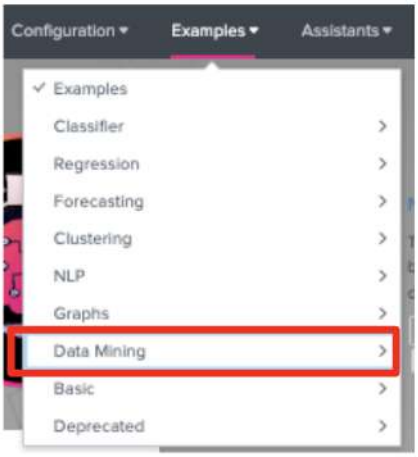
In this task you will review an anomaly detection modeling example in the DSDL. Take a moment to observe the visualization of both the outlier data and the predicted outliers

Steps

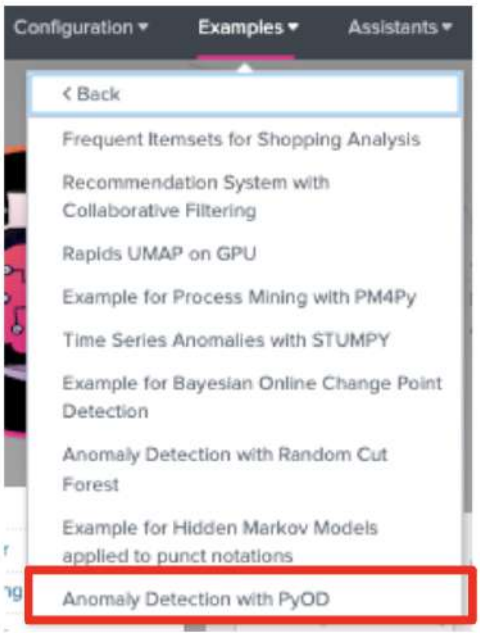
1. Go back to your original Tab and click on “Examples”



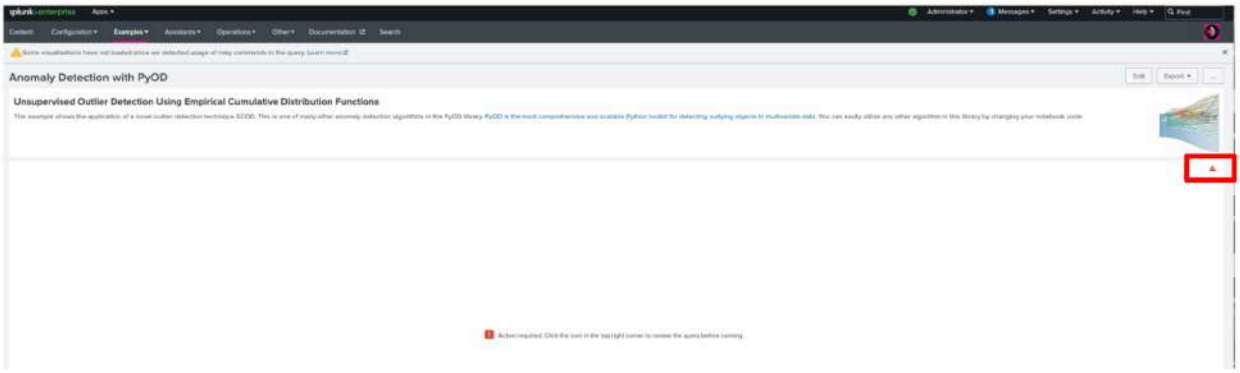
2. Select **Data Mining**



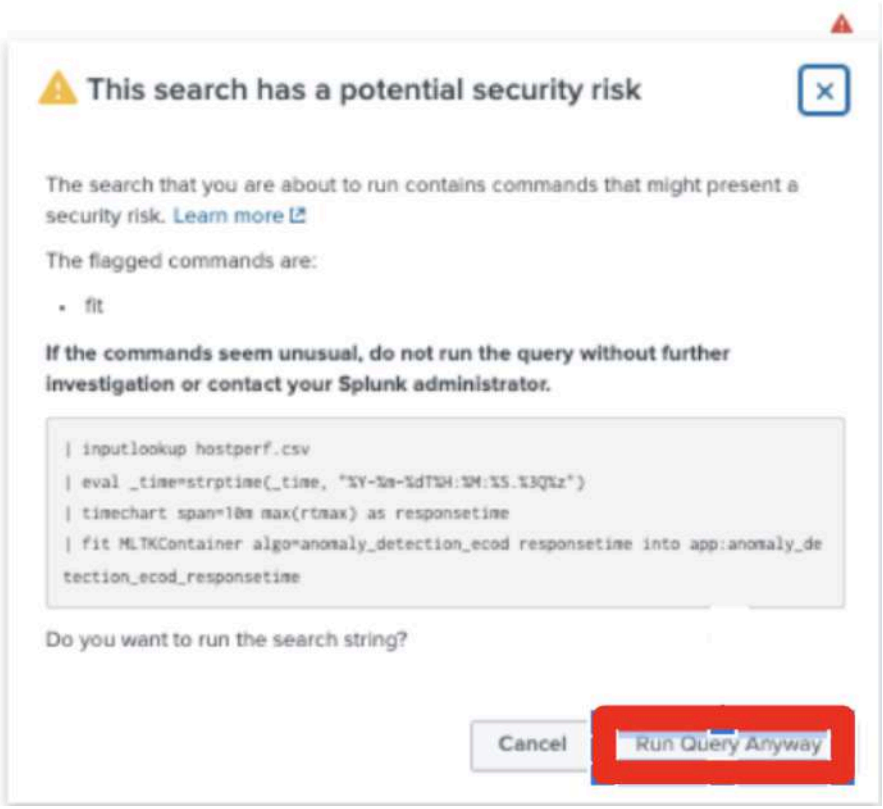
3. Then select **Anomaly Detection with PyOD**



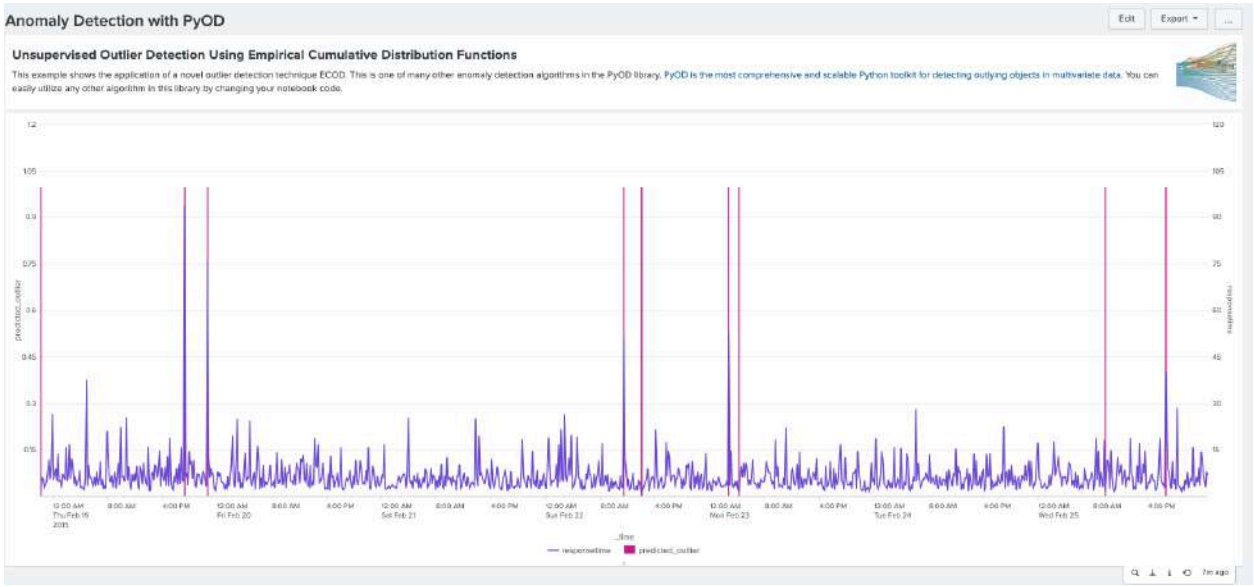
4. Click the warning triangle



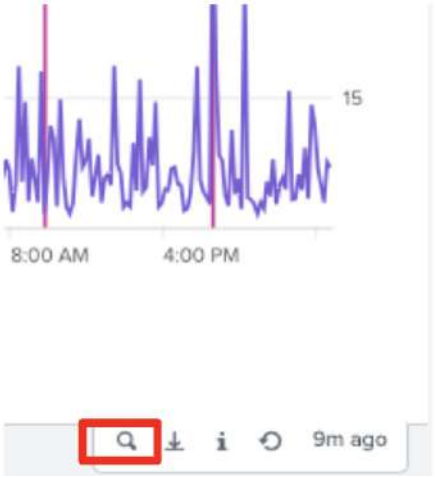
5. Click **Run Query Anyway**



6. We just used the “PyOD” library to perform an unsupervised outlier detection. Notice the blue line is mapping the response time outliers within the dataset and the red lines represent predicted outliers.



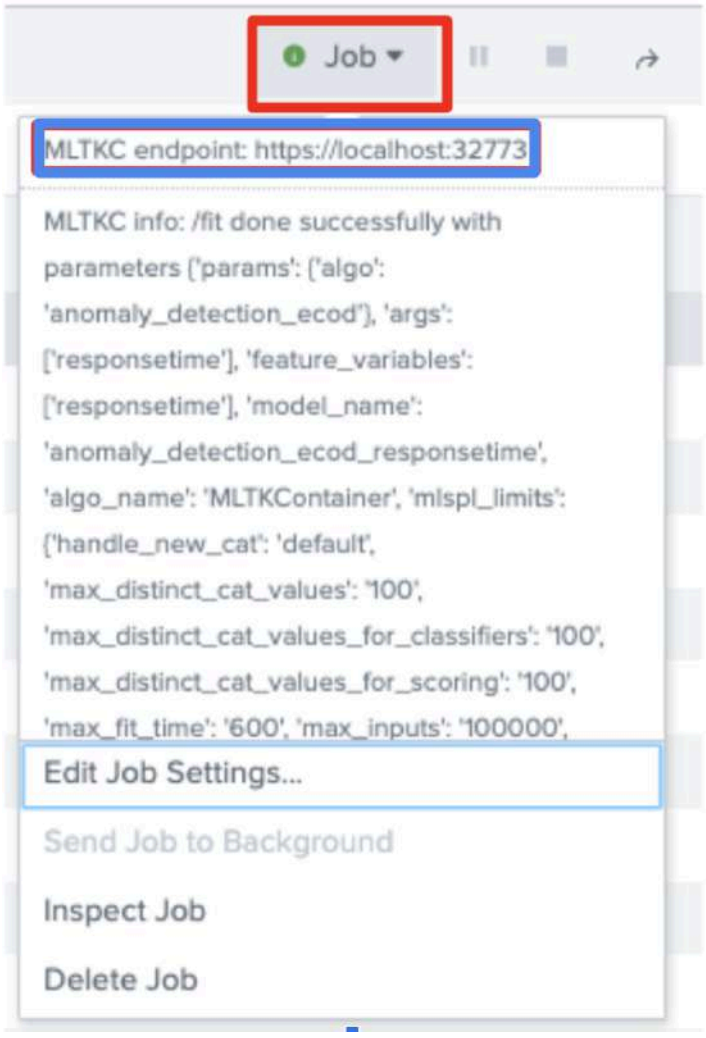
7. Let's take a closer look at our SPL that generated this output. Click the magnifying glass *(Make sure to hover over the panel for the magnifying glass to appear)*



8. A new search tab will open up and you can see the SPL and the results that were graphed in addition to the prediction scores. Notice the SPL contains a **fit** command. This command includes a setting to use your MLTKContainer. You will also notice the algorithm name is *anomaly_detection_ecod*

Click on the sort areas in the predicted_outlier column to see how many outliers were identified.

9. Next click on the **Job** down arrow. You will see the job has been sent to the container for processing. Remember, one purpose of the DSDL is to offload the modeling process to the containerized compute environment.



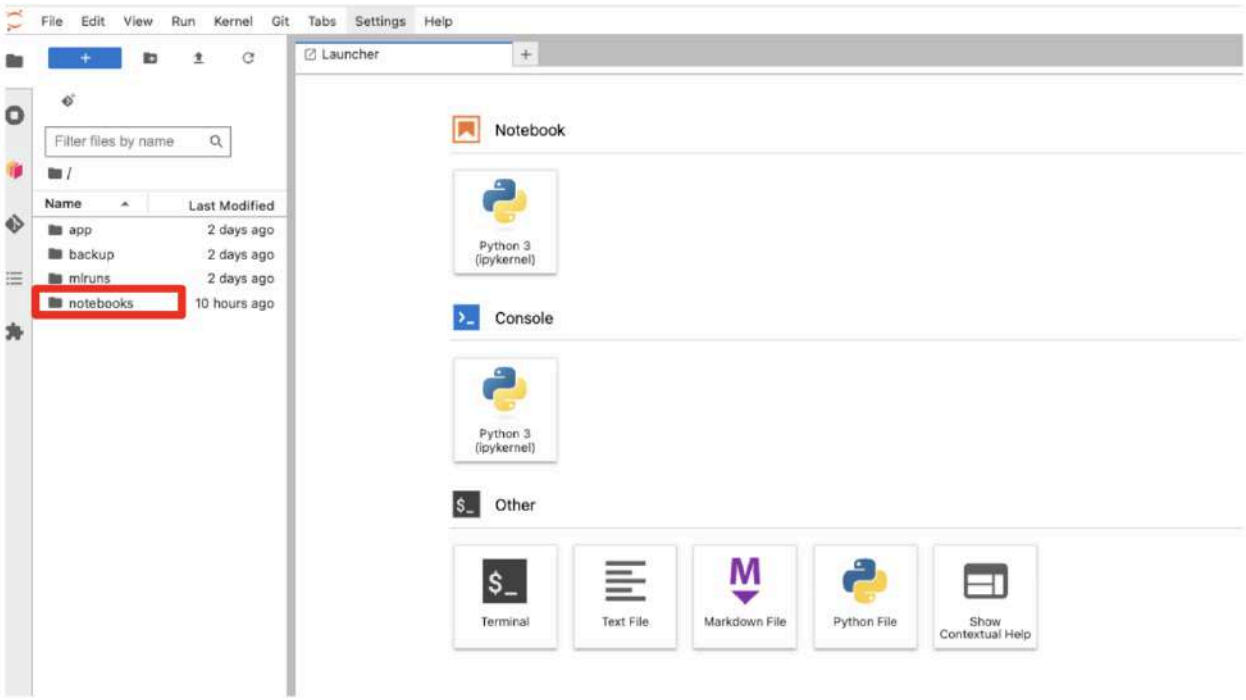
Now that you've seen an example of what is possible, it is now time to see how the example was created. Over the remaining tasks you will recreate what you just viewed in the DSDL example.

Task 3: Access the Jupyter Lab

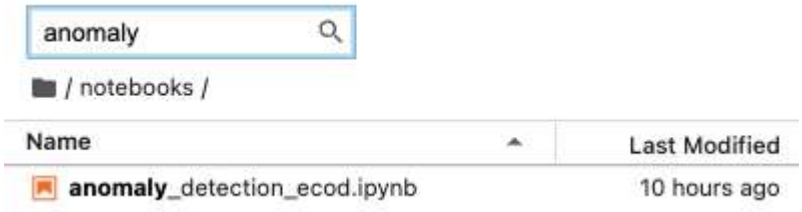
In this scenario you are using Splunk's integration with Jupyter Notebooks to create the custom model you are using in your SPL. The Jupyter Lab allows you to perform the data analysis, feature engineering, and model initialization and training with the Jupyter Notebook. Let's take a look at the example Jupyter Notebook.

Steps

1. Go to the Jupyter Environment tab you created earlier and double click on **notebooks**



2. Do a search for *anomaly* and you will find the *anomaly_detection_ecod* notebook. Double click it.



3. In the following steps, you are going to run each cell of python code in the notebook. When you want to run a cell, click on the cell you want to run and click the run icon

Or press **shift** + **return** on your keyboard

4. Run each of the following cells in *1.2. Stage 0 - import libraries*:
    a. Import your libraries

```
[2]: # this definition exposes all python module imports that should be available in all subsequent commands
import json
import numpy as np
import pandas as pd
from pyod.models.ecod import ECOD
#from pyod.models.iforest import IForest
# ...
# global constants
MODEL_DIRECTORY = "/srv/app/model/data/"
```

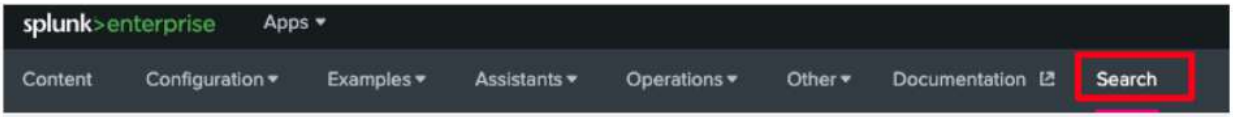
    b. Verify your library versions

```
[3]: # THIS CELL IS NOT EXPORTED - free notebook cell for testing or development purposes
print("numpy version: " + np.__version__)
print("pandas version: " + pd.__version__)

numpy version: 1.26.4
pandas version: 2.2.2
```

    c. Verify the algorithm you will use is available and has metadata. Feel free to scroll through the output to familiarize yourself with the data structure used by the model.

```
[4]: dir(ECOD) # Model Metadata

'__str__',
'__subclasshook__',
'__weakref__',
'_abc_impl',
'_decision_function_parallel',
'_get_param_names',
'_predict_rank',
'_process_decision_scores',
'_set_n_classes',
'decision_function',
'explain_outlier',
'fit',
'fit_predict',
'fit_predict_score',
'get_params',
'predict',
'predict_confidence',
'predict_proba',
```

5. Before you move forward to the next step, you should familiarize yourself with the dataset we will be working with. Click on your Splunk UI browser tab and open search



6. Next, run the following search to view the entire dataset you will be working with.

```
| inputlookup hostperf.csv
```

Take note of the number of results in the dataset and the abbreviated response time values. You will use Splunk to create a sample from the dataset and create a new field (*responsetime*), which will be the feature used to train your model.

Task 4: Test Sample Dataset Creation

Now that you are familiar with the dataset, it is time to create your sample dataset you will use for modeling. You will establish a connection to Splunk and use a search to pull a dataset directly from Splunk for sample dataset creation. You will first adjust the search to generate the sample size you want and then you will add a **fit** command to generate the final sample dataset to be used for modeling.

Steps

1. Click on the Jupyter notebook browser tab and scroll to *1.3. Stage 1 - get a data sample from Splunk* and then go to *1.3.1.1. Option 1.a - Create test sample dataset.* For the next steps you will be using the cells in this section.

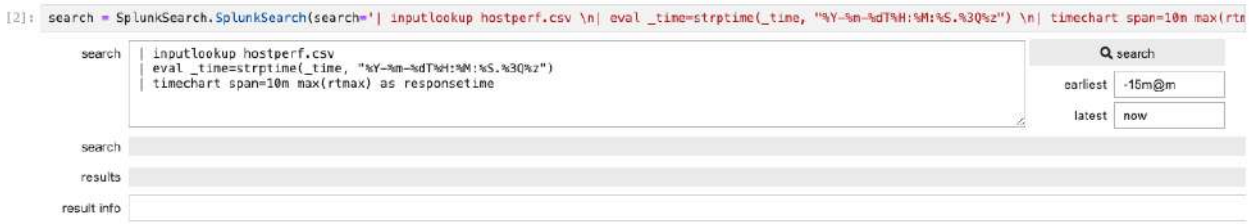
2. Run the first cell to import the Splunk library which is needed to communicate with your splunk environment.

```
[6]: #interact with Splunk to validate your search results.  Change span to 30m
     from dsdlsupport import SplunkSearch as SplunkSearch
```

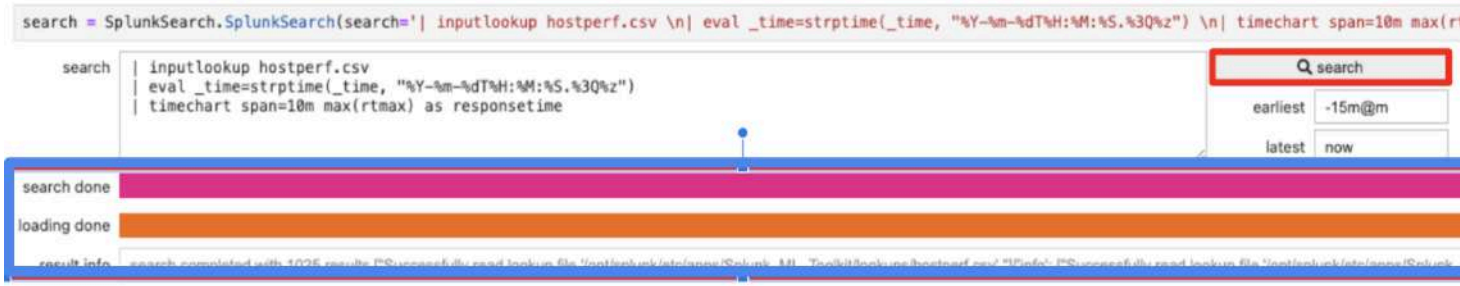
3. Next, run the next cell to establish the search connection with Splunk

```
search = SplunkSearch.SplunkSearch(search='| inputlookup hostperf.csv \n| eval _time=strptime(_time, "%Y-%m-%dT%H:%M:%S.%3Q%z") \n| timechart span=10m max(rtm
```

4. Once the connection is established, run the next cell to initialize a Splunk search window. The window will appear in the cell as shown below.



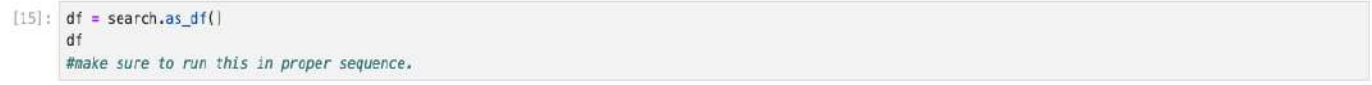
5. Click the search button and wait for the status bars and results info to show completed.
   IMPORTANT: Do not click “run” after this has completed.



6. Next, you will ensure the dataframe has been populated with the search results.

   Click on the next cell and then click **run**.

   Note:The previous cell must still show completed status bars in order for the dataframe to be populated with results.



	_time	responsetime	_span
0	2015-02-18T22:10:00.000+00:00	1.275	600
1	2015-02-18T22:20:00.000+00:00	5.933	600
2	2015-02-18T22:30:00.000+00:00	5.599	600
3	2015-02-18T22:40:00.000+00:00	2.839	600
4	2015-02-18T22:50:00.000+00:00	3.702	600
...	...	...	...
1020	2015-02-26T00:10:00.000+00:00	4.894	600
1021	2015-02-26T00:20:00.000+00:00	2.898	600
1022	2015-02-26T00:30:00.000+00:00	7.564	600
1023	2015-02-26T00:40:00.000+00:00	6.938	600
1024	2024-05-17T18:40:00.000+00:00	NaN	600

1025 rows × 3 columns

*If the dataframe does not return results, rerun the search and ensure the results are showing successful as pictured above in **Task 4: step 5**. Then rerun the dataframe cell.

Notice the number of results we have generated for our sample dataset. You will also notice you have
enriched your data with a new field (*responsetime*) that will be used for modeling results.

7. You can change the span parameter as needed to adjust the size of the sample dataset. *Sample size can impact the accuracy of your modeling results. It is best to try multiple sample sizes to optimize your modeling results.*

   Before you do, make sure to rerun the “SplunkSearch” import step first.

```
#interact with Splunk to validate your search results.  Change span to 30m
from dsdlsupport import SplunkSearch as SplunkSearch
```

   This ensures your dataframe is populated with your new search results.

8. Now, try a few span iterations and observe how they impact your sample size. The example below shows a change from **span=10m** to **span=20m** *For this step, please set span to 2m or greater when testing.*


```
df = search.as_df()
df
#make sure to run this in proper sequence.
```

	_time	responsetime	_span
0	2015-02-18T22:00:00.000+00:00	1.275	1200
1	2015-02-18T22:20:00.000+00:00	5.933	1200
2	2015-02-18T22:40:00.000+00:00	3.702	1200
3	2015-02-18T23:00:00.000+00:00	8.361	1200
4	2015-02-18T23:20:00.000+00:00	11.885	1200
...	...	...	...
509	2015-02-25T23:40:00.000+00:00	14.206	1200
510	2015-02-26T00:00:00.000+00:00	6.892	1200
511	2015-02-26T00:20:00.000+00:00	7.564	1200
512	2015-02-26T00:40:00.000+00:00	6.938	1200
513	2024-05-20T13:00:00.000+00:00	NaN	1200

514 rows × 3 columns

*Notice the change in the sample size. You can repeat these steps until you have the sample size you want. When you are have decided which span setting you will use, move on to Task 5. The remaining steps will be using **span=10m***

Task 5: Final Sample Dataset Creation

It is time to create the dataset you will use for modeling. Now that you have the search that will be used to create your dataset, you will need to collect the features(fields) you will be modeling for and send this data over to the container for modeling. To accomplish this you will use the **fit** command.

The **fit** command is used to produce a machine learning model based on the behavior of a set of events and applies the machine learning model to the current set of search results in the search pipeline. Running the fit command by default updates the model, which is what we don't want you to do in this example. You will be using a special setting (**mode=stage**) with the **fit** command to accomplish this.

The **mode=stage** setting does not actually fit the data or modify the model. This setting indicates that you want to transfer a dataset over to the container to be worked on in the JupyterLab environment.

Notice the command below uses the **mode=stage**. It designates the algorithm**(***anomaly_detection_ecod*) to use and the feature(*responsetime*) to model.

Note: You created the field 'responsetime" using the eval command in your SPL search

```
Unset
| fit MLTKContainer mode=stage algo=anomaly_detection_ecod responsetime into
app:anomaly_detection_ecod_responsetime
```

Steps

1. Scroll to *Option 1.b - Create final sample dataset*

```
#### Option 1.b - Create final sample dataset
```

2. Run the first cell to re-Initialize your "SplunkSearch" connection..

```
#interact with Splunk to validate your search results.  Create final sample dataset
from dsdlsupport import SplunkSearch as SplunkSearch
```

3. Copy the new search with the **fit** command from the code block below

```
search = SplunkSearch.SplunkSearch(search='| inputlookup hostperf.csv \n| eval
_time=strptime(_time, "%Y-%m-%dT%H:%M:%S.%3Q%z") \n| timechart span=10m max(rtmax) as
responsetime \n| fit MLTKContainer mode=stage algo=anomaly_detection_ecod
responsetime into app:anomaly_detection_ecod_responsetime')
```

…and click on the next cell, as shown below, and paste the new search into the cell. Then click **run**.

```
#add new search here
```

*Note: The **span** in this example has been changed back to 10m for this final sample dataset which will produce a sample dataset with 1026 rows, as you will see below. Depending on which span definition you use, your sample dataset size may contain a different number of rows.*

4. Once the new search window is initialized, click **search** to create the final dataset.

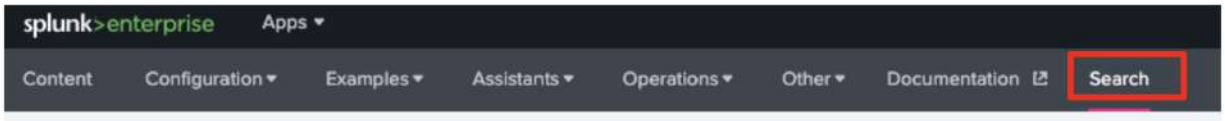
5. Next, run the dataframe cell to ensure the dataframe is populated. *Remember the status bars above most show completed.*

```
df = search.as_df()
```

	_time	responsetime	_span
0	2015-02-18T22:10:00.000+00:00	1.275	600
1	2015-02-18T22:20:00.000+00:00	5.933	600
2	2015-02-18T22:30:00.000+00:00	5.599	600
3	2015-02-18T22:40:00.000+00:00	2.839	600
4	2015-02-18T22:50:00.000+00:00	3.702	600
...	...	...	...
1021	2015-02-26T00:20:00.000+00:00	2.898	600
1022	2015-02-26T00:30:00.000+00:00	7.564	600
1023	2015-02-26T00:40:00.000+00:00	6.938	600
1024	2024-05-20T13:30:00.000+00:00	NaN	600
1025	2024-05-20T13:40:00.000+00:00	NaN	600

1026 rows × 3 columns

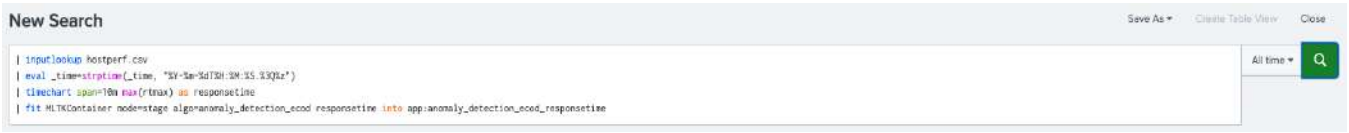
6. To support use cases where dataset generation needs to be initiated directly from Splunk and not from a Jupyter Notebook(e.g., operationalized dataset generation by roles with no direct Jupyter Notebooks access), you can also push the data to the Jupyter Notebook from Splunk. Click on your Splunk tab in the browser. Then click on “Search”



7. Using the same search, we used in the Jupyter notebook to create the sample dataset

```
| inputlookup hostperf.csv
| eval _time=strptime(_time, "%Y-%m-%dT%H:%M:%S.%3Q%z")
| timechart span=10m max(rtmax) as responsetime
| fit MLTKContainer mode=stage algo=anomaly_detection_ecod responsetime into
app:anomaly_detection_ecod_responsetime
```

Run the search directly in Splunk



8. Your results will look similar

9. Once completed, go back to your Jupyter Lab tab in your browser

10. Scroll to section *1.3.2. Option 2 - push data from Splunk,* and run the first two cells

```
# this cell is not executed from MLTK and should only be used for staging data into the notebook environment
def stage(name):
    with open("data/"+name+".csv", 'r') as f:
        df = pd.read_csv(f)
    with open("data/"+name+".json", 'r') as f:
        param = json.load(f)
    return df, param
```

```
# THIS CELL IS NOT EXPORTED - free notebook cell for testing or development purposes
df, param = stage("anomaly_detection_ecod_responsetime")
```

These cells save your dataset to a .csv file and your metadata to a json file on your MLTKC container. It will then populate a dataframe and a param variable with the dataset and metadata respectively. These will be used within the Jupyter Notebook for the following modeling tasks.

11. Now, run the next two cells. You should see similar results.

```
df
```

[20]:

	responsetime
0	1.275
1	5.933
2	5.599
3	2.839
4	3.702
...	...
1019	6.892
1020	4.894
1021	2.898
1022	7.564
1023	6.938

1024 rows × 1 columns

```
[21]: param
```

```
[21]: {'options': {'params': {'mode': 'stage', 'algo': 'anomaly_detection_ecod'},
   'args': ['responsetime'],
   'feature_variables': ['responsetime'],
   'model_name': 'anomaly_detection_ecod_responsetime',
   'algo_name': 'MLTKContainer',
   'mlspl_limits': {'handle_new_cat': 'default',
    'max_distinct_cat_values': '100',
    'max_distinct_cat_values_for_classifiers': '100',
    'max_distinct_cat_values_for_scoring': '100',
    'max_fit_time': '600',
    'max_inputs': '100000',
    'max_memory_usage_mb': '4000',
    'max_model_size_mb': '30',
    'max_score_time': '600',
    'use_sampling': 'true'},
   'kfold_cv': None},
  'feature_variables': ['responsetime']}
```

Task 6: Modeling

Now that you have created a final sample data set,, it is time to create your model. Remember modeling processing is offloaded to the containerized environment.

In this step we will initialize your model, train, test, and save your model.

Note: In a normal model creation process, you would perform an iterative process of training, inspecting, validating, testing, and retraining your model to improve your model via code enhancement or new data sets. Since this is an interactive workshop, we will bypass those steps.

Steps

1. Scroll to the *1.4. Stage 2 - create and initialize a model section*
2. Run the following two cells to initialize your model with the dataset and metadata.

```
# initialize your model
# available inputs: data and parameters
# returns the model object which will be used as a reference to call fit, apply and summary subsequently
def init(df,param):
    model = ECOD(contamination=0.01)
    # parallization options for ECOD:
    # ECOD(n_jobs=2)
    # most of other PyOD models would work similar, e.g. replace with Isolation Forest:
    # model = IForest()

    return model
```

```
# THIS CELL IS NOT EXPORTED - free notebook cell for testing or development purposes
model = init(df,param)
print(model)
```

```
ECOD(contamination=0.01, n_jobs=1)
```

3. Now that your model is initialized, it is time for you to train the model using the **fit** command. Scroll to section *1.5. Stage 3 - fit the model*, and run the first cell, which uses python code to train your model.

```
# train your model
# returns a fit info json object and may modify the model object
def fit(model,df,param):
    X = df[param['feature_variables'][0]]
    X_train = np.reshape(X.to_numpy(), (len(X), 1))

    # contamination = 0.01
    model.fit(X_train)

    info = {"message": "model trained"}
    return info
```

```
# THIS CELL IS NOT EXPORTED - free notebook cell for testing or development purposes
print(fit(model,df,param))
```

```
{'message': 'model trained'}
```

Once complete, make sure you see the message “model trained”

Note: The benefit is you can utilize the full capabilities of python to train your customized models which includes libraries and algorithms that aren’t included in the Splunk MTLK. These models can be operationalized directly from Splunk searches or exported as pretrained ONNX models and imported into the MLTK for further operationalization..

4. Now that your model is trained. It is time to use the **apply** command to run the model with your test dataset and get the results.

   The **apply** command is used to apply the machine learning model that was learned using the **fit** command. It repeats a selection of the **fit** command steps

   Note: this is an anomaly detection model, so you will look for outlier values in the results.

   Scroll to the *1.6. Stage 4 - apply the model* section.

5. Run the cell to perform the **apply** command using your trained model and your dataset

```
# apply your model
# returns the calculated results
def apply(model,df,param):
    X = df[param['feature_variables']][0]]
    X_apply = np.reshape(X.to_numpy(), (len(X), 1))

    y_hat = model.predict(X_apply)  # outlier labels (0 or 1)
    y_scores = model.decision_function(X_apply)  # outlier scores

    result = pd.DataFrame(y_hat, columns=['outlier'])
    return result
```

6. Run the next cell to view the results of the model

```
# THIS CELL IS NOT EXPORTED - free notebook cell for testing or development purposes
print(apply(model,df,param))
```

Notice the results include an outlier column

```
      outlier
0           1
1           0
2           0
3           0
4           0
...       ...
1019        0
1020        0
1021        0
1022        0
1023        0

[1024 rows x 1 columns]
```

7. Proceed to run the remaining cells to save, load, and summarize the model

1.7. Stage 5 - save the model

```
# save model to name in expected convention "<algo_name>_<model_name>"
# for model persistence you can save the model. This is optional
def save(model,name):
    if model is not None:
        if isinstance(model,ECOD):
            from joblib import dump, load
            dump(model, MODEL_DIRECTORY + name + '.joblib') #model directory is on the container side
    return model
```

1.8. Stage 6 - load the model

```
# load model from name in expected convention "<algo_name>_<model_name>"
def load(name):
    model = {}
    from joblib import dump, load
    model = load(model, MODEL_DIRECTORY + name + '.joblib')
    return model
```

1.9. Stage 7 - provide a summary of the model

```
# return a model summary
def summary(model=None):
    returns = {"version": {"numpy": np.__version__, "pandas": pd.__version__} }
    return returns
```

Task 7: Review Results and Visualization Creation

Now that you have created your model, you will perform the next steps to quickly operationalize the model.

Visualizations are essential to operationalizing the results from your modeling process. You can use these visualizations to populate panels in your custom Splunk dashboards which will make your results easily consumable.

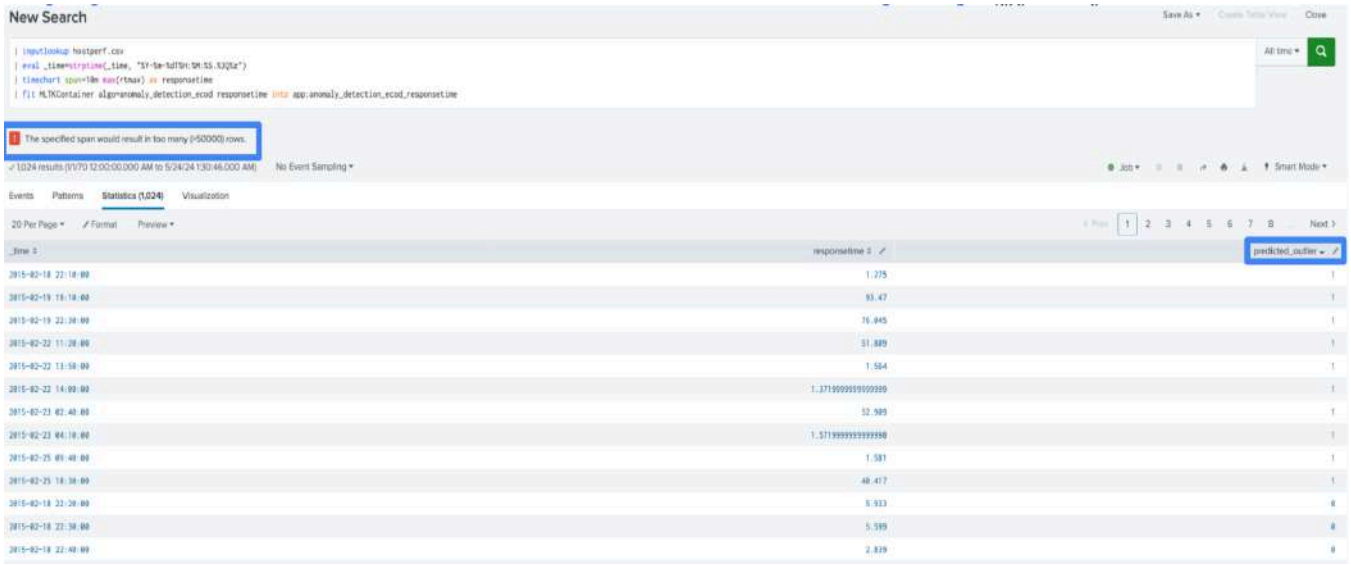
In this final task, you will use your model within the Splunk UI to generate search results and create a visualization.

Steps

1. Now click the Splunk UI browser tab and run the following search. Notice we have removed the **mode=stage** parameter.

```
| inputlookup hostperf.csv
| eval _time=strptime(_time, "%Y-%m-%dT%H:%M:%S.%3Q%z")
| timechart span=10m max(rtmax) as responsetime
| fit MLTKContainer algo=anomaly_detection_ecod responsetime into
app:anomaly_detection_ecod_responsetime
```

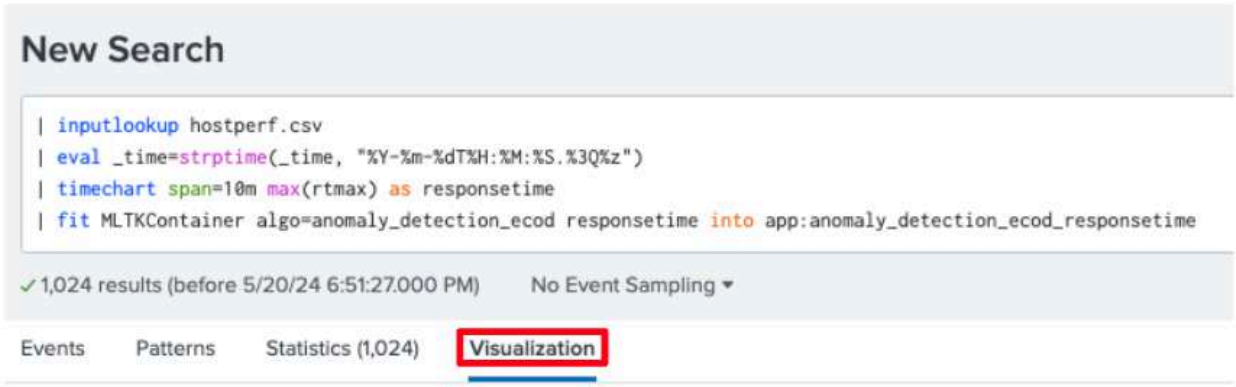
2. Your results should look similar. Take a moment to sort the *Outlier* column to see how many were identified.



Note: You may receive a "The specified span would result in too many (>50000) rows." warning. You can ignore this message for the purpose of the workshop.

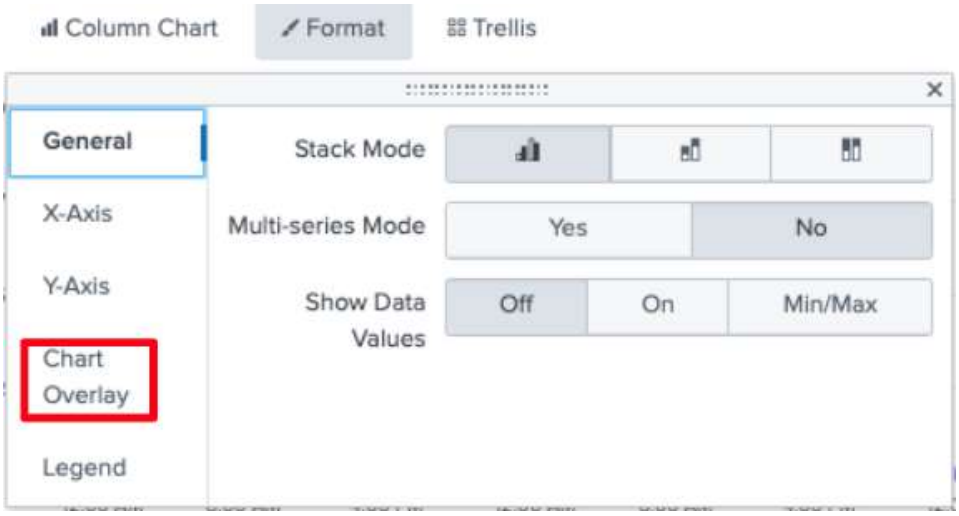
3. You will now re-create the example visualization from *Task 2: step 6*.

   Click on **visualization**

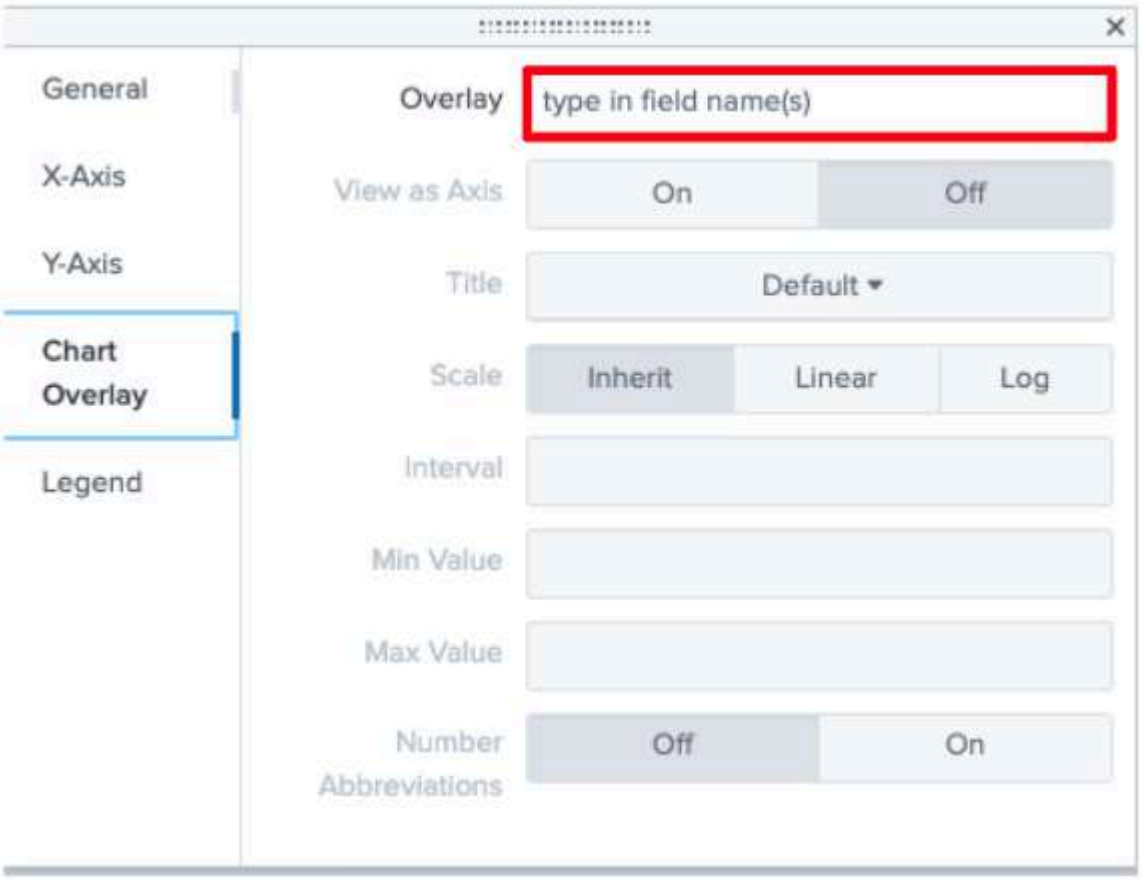


4. And then **Format**

5. Then click on **Chart Overlay**

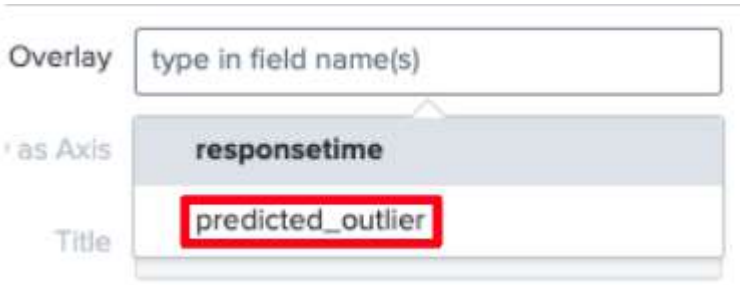


6. Next, click inside the **Overlay** input field to activate the selection drop-down



splunk>

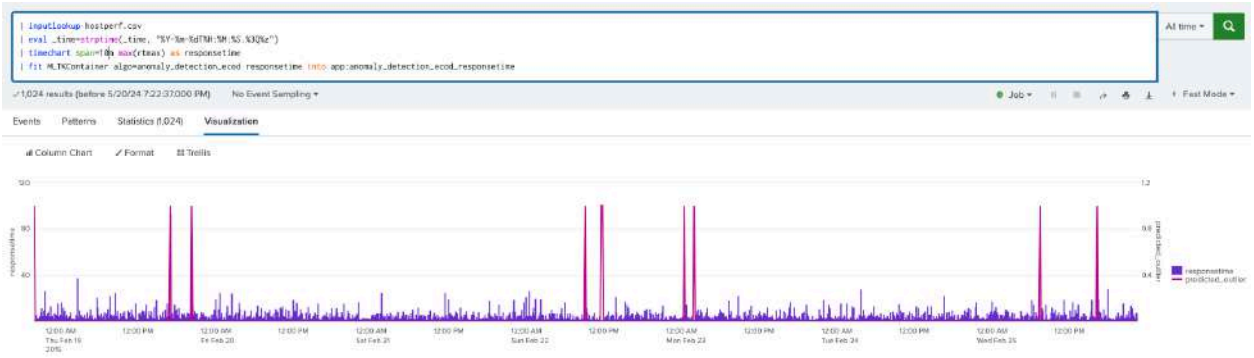
7. …and select **predicted_outlier**



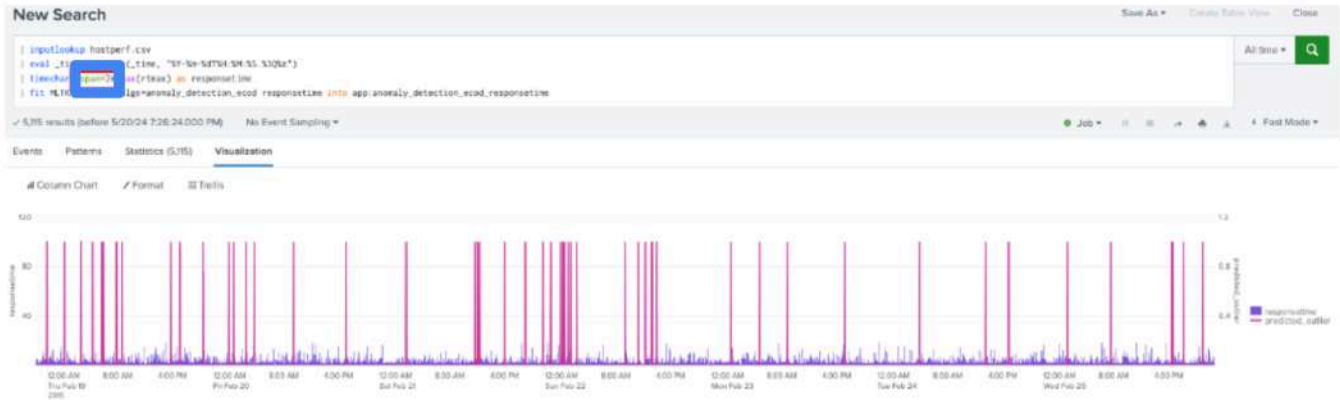
8. Next click the **On** button for **View as Axis**

…and close the pop-up window.

9. You have now created a new visualization using the model you've created in the DSDL.



10. Try adjusting the "span=". We recommend nothing less than 2m for this dataset.



splunk>

In this module, you were introduced to some of the capabilities of the Splunk Data Science and Deep Learning (DSDL) app (e.g., Containerization, Jupyter/Splunk integration, custom model training & deployment, and model operationalization via dashboard visualization).

Remember, the Data Science and Machine Learning (DSDL) app extends the Splunk Machine Learning Toolkit (MLTK) with prebuilt Docker containers for TensorFlow, PyTorch, and a collection of data science, NLP, and classical machine learning libraries. By using predefined workflows for rapid development with Jupyter Lab Notebooks, the DSDL enables you to build, test,and operationalize your custom models with Splunk. For your process heavy modeling requirements, the DSDL enables you to leverage GPUs for compute intense training tasks and flexibly deploy models on CPU or GPU enabled containers. Use the DSDL to perform custom AI modeling or to create pretrained ONNX models to upload into the MLTK for further operationalization.