Splunk Machine Learning Toolkit (MLTK)

The Splunk Machine Learning Toolkit (MLTK) is an app for Splunk Enterprise and Splunk Cloud that lets you apply machine learning techniques to your machine data without needing to be a data scientist. It integrates directly with Splunk's search processing language (SPL) and provides guided workflows, assistants, and custom commands to build, test, and deploy ML models.

Key Features of Splunk MLTK

- ML Assistants – Prebuilt guided workflows for regression, classification, anomaly detection, clustering, and forecasting.
- SPL Integration – Provides custom SPL commands such as fit (train), apply (apply models), summary (summarize models), and score (evaluate results).
- Algorithms & Libraries – Built on scikit-learn, statsmodels, and Splunk's ML library.
- Model Management – Save, load, and reuse trained ML models inside Splunk.
- Visualization & Evaluation – Supports accuracy metrics and dashboards for predictions and anomalies.

Typical Use Cases

- Anomaly detection – Find unusual spikes/drops in system metrics or logs.
- Predictive analytics – Forecast system resource utilization, network traffic, or sales data.
- Classification – Detect fraud, categorize alerts, or classify log events.
- Clustering – Group similar events (e.g., by error codes, transaction types).
- Forecasting – Predict future values in time-series data.

Example SPL Commands

Train a regression model:

```
| inputlookup cpu_usage.csv
| fit LinearRegression "cpu_load" from "time","host" into cpu_model
```

Apply the saved model:

```
| inputlookup new_cpu_data.csv
| apply cpu_model
```

Detect anomalies:

```
| inputlookup web_traffic.csv
| anomalydetection method=stl field=requests
```

Installation & Requirements

- Available from Splunkbase (free app).
- Requires Python for Scientific Computing (PSC) add-on.
- Works with Splunk Enterprise and Splunk Cloud.

◇ 1. Prerequisites

- **Splunk Enterprise or Splunk Cloud** account
- Admin privileges to install apps
- **Python for Scientific Computing (PSC)** add-on (needed for ML algorithms)
- Internet access to Splunkbase (or download package manually if offline)

◇ 2. Install Python for Scientific Computing (PSC)

Before MLTK, you must install the PSC add-on:

1. Go to **Splunkbase** → Search for *"Python for Scientific Computing"*.
   - For **Linux**: PSC add-on for Linux
   - For **Windows**: PSC add-on for Windows
   - For **macOS**: PSC add-on for macOS
2. Download the appropriate package.
3. In **Splunk Web**:
   - Go to **Apps ▸ Manage Apps ▸ Install app from file**.
   - Upload the PSC `.tar.gz` package.
4. Restart Splunk after installation.

◇ 3. Install Machine Learning Toolkit (MLTK)

1. Log in to **Splunk Web** as an Admin.
2. Navigate to:
   Apps ▸ Find More Apps.
3. Search for **Machine Learning Toolkit**.
4. Click **Install**.

- If Splunkbase credentials are required, enter them.

5. Restart Splunk after installation.

◇ 4. Verify Installation

- Go to **Apps menu** → You should see **Machine Learning Toolkit**.
- Open it and check that **ML Assistants** (Forecasting, Anomaly Detection, Predict Numeric Fields, etc.) are available.
- If missing, ensure PSC add-on is installed correctly.

◇ 5. (Optional) Install Example Datasets

Splunk provides a companion app **"MLTK Examples"**:

- Contains sample data, prebuilt dashboards, and example use cases.
- Useful for training & practice.

You can install it the same way via **Apps ▸ Find More Apps**.

1) Using the MLTK "Predict Numeric Fields" Assistant (GUI)

1. **Open**: Apps → **Machine Learning Toolkit** → **Predict Numeric Fields**.
2. **Choose Data**:
   - Search example (adapt to your index/sourcetype):

```
3. index=infra_metrics sourcetype=perf:host host=*
4. | timechart span=1m avg(cpu_user) avg(cpu_system) avg(mem_used)
   avg(net_in) avg(net_out) by host
5. | rename "avg(cpu_user)" as cpu_user "avg(cpu_system)" as
   cpu_system "avg(mem_used)" as mem_used "avg(net_in)" as net_in
   "avg(net_out)" as net_out
6. | eval cpu_load=cpu_user+cpu_system
7. | fields _time host cpu_load cpu_user cpu_system mem_used net_in
   net_out
```

8. **Target & Features**:
   - Target field: `cpu_load`
   - Feature fields: `cpu_user cpu_system mem_used net_in net_out`
9. **Algorithm**: **LinearRegression**.
10. **Validation**: enable **k-fold CV** (e.g., 5) or **Holdback** (e.g., 20%).
11. **Train** → review metrics (R^2, RMSE) & **residuals plot**.
12. **Save Model**: give a name like `mltk_cpu_load_lr`.

13. **Operationalize**: click **Show SPL**, copy the `| fit … into mltx_model` and `| apply …` searches to reuse in alerts/dashboards.

---

2) Pure SPL Workflow (reusable in searches/jobs)

A. Data prep (clean & select features)

```
index=infra_metrics sourcetype=perf:host host=*
| bin _time span=1m
| stats avg(cpu_user) as cpu_user avg(cpu_system) as cpu_system
avg(mem_used) as mem_used avg(net_in) as net_in avg(net_out) as net_out
by _time host
| eval cpu_load = cpu_user + cpu_system
| fillnull value=0 cpu_user cpu_system mem_used net_in net_out cpu_load
| fields _time host cpu_load cpu_user cpu_system mem_used net_in
net_out
```

B. Train a Linear Regression model (with cross-validation)

```
...  <!-- (append to the search above) -->
| fit LinearRegression "cpu_load" from "cpu_user" "cpu_system"
"mem_used" "net_in" "net_out"
    kfold_cv=5
    into m_cpu_load_lr
```

- `kfold_cv=5` does 5-fold cross-validation.
- Use `holdback=0.2` instead if you prefer a 20% test split.

C. Inspect the trained model (coefficients & details)

```
| summary m_cpu_load_lr
```

D. Apply the saved model to new data

```
index=infra_metrics sourcetype=perf:host host=* earliest=-15m
| bin _time span=1m
| stats avg(cpu_user) as cpu_user avg(cpu_system) as cpu_system
avg(mem_used) as mem_used avg(net_in) as net_in avg(net_out) as net_out
by _time host
| eval cpu_load = cpu_user + cpu_system
| fillnull value=0 cpu_user cpu_system mem_used net_in net_out cpu_load
| apply m_cpu_load_lr as predicted_cpu_load
```

E. Evaluate accuracy (RMSE, MAE, R^2 via SPL)

```
...  <!-- after apply -->
| eval residual = cpu_load - predicted_cpu_load
| eval se = pow(residual, 2), ae = abs(residual)
| eventstats avg(se) as MSE avg(ae) as MAE
| eval RMSE = sqrt(MSE)
| stats first(RMSE) as RMSE first(MAE) as MAE \
       corr(cpu_load, predicted_cpu_load) as PearsonR
| eval R2 = pow(PearsonR, 2)
```

(If you prefer, you can also use MLTK's `score` command; the above SPL is explicit and dashboard-friendly.)

---

3) Handling categorical fields (quick one-hot approach)

If you have a categorical field like `env` with values `prod|qa|dev`, create dummy variables:

```
| eval env_prod = if(env="prod",1,0), env_qa = if(env="qa",1,0),
env_dev = if(env="dev",1,0)
| fit LinearRegression "cpu_load" from "cpu_user" "cpu_system"
"mem_used" "net_in" "net_out" "env_prod" "env_qa" "env_dev" into
m_cpu_load_lr
```

---

4) Scheduling & dashboards

Scheduled scoring search (every 5 minutes) to populate a summary index:

```
index=infra_metrics sourcetype=perf:host earliest=-6m latest=now
| bin _time span=1m
| stats avg(cpu_user) as cpu_user avg(cpu_system) as cpu_system
avg(mem_used) as mem_used avg(net_in) as net_in avg(net_out) as net_out
by _time host
| eval cpu_load = cpu_user + cpu_system
| fillnull value=0 *
| apply m_cpu_load_lr as predicted_cpu_load
| eval residual = cpu_load - predicted_cpu_load
| collect index=ml_predictions source="cpu_lr" marker="cpu_linear"
```

Simple dashboard panel (sparkline of residuals):

```
index=ml_predictions source=cpu_lr
| timechart span=1m avg(residual) as avg_residual by host
```

---

5) Tips, gotchas & best practices

- **Feature scaling**: Linear regression doesn't require scaling, but large-magnitude features can dominate numeric stability; consider normalizing inputs when ranges vary wildly.
- **Multicollinearity**: Highly correlated features (e.g., `cpu_user` and `cpu_system`) can inflate variance of coefficients; try removing one or switch to **Ridge**/**Lasso** (also available in MLTK) if stability is an issue:

- `| fit Ridge "cpu_load" from cpu_user cpu_system mem_used net_in net_out alpha=1.0 into m_cpu_load_ridge`
- **Drift**: Re-train periodically if relationships change (e.g., monthly) and keep old models versioned (`m_cpu_load_lr_v1`, `v2`, …).
- **Data quality**: Handle nulls/outliers before training; use `fillnull`, winsorization logic, or conditional filtering.
- **Explainability**: Use `summary` to view coefficients; large absolute coefficients indicate stronger influence (subject to scaling).
- **Access**: Save models in an app context accessible to search heads that will **apply** them (especially in SHC/Cloud).

1) The four workhorse commands

A. `fit`

Trains and (optionally) persists a model.

Shape

```
| fit LinearRegression <target_field>
      from <feature_1> <feature_2> ...
      [into <model_name>]
      [kfold_cv=<int> | holdback=<0-1>]
      [partition_by=<field>]
      [fit_intercept=<true|false>]
      [normalize=<true|false>]
```

Notes

- `kfold_cv` does k-fold cross-validation; `holdback` reserves a random fraction for testing (use one or the other).
- `partition_by` trains **one model per category** (e.g., per `host`, `app`, `region`).
- `fit_intercept` (aka include intercept) and `normalize` map to the underlying learner's params.
- If you omit `into`, the model is trained in-memory for that search only. Use `into <name>` to save and reuse.

Examples

```
... | fit LinearRegression price from sqft bedrooms bathrooms into m_price_lr

... | fit LinearRegression cpu_load from cpu_user cpu_system mem_used \
```

```
      kfold_cv=5 into m_cpu_lr

... | fit LinearRegression latency from rps payload_kb conn \
      holdback=0.2 fit_intercept=true normalize=false \
      partition_by=service into m_latency_by_svc
```

B. `apply`

Loads a saved model and produces predictions.

Shape

```
| apply <model_name> [as <pred_field>]
```

Notes

- Feature **field names must exist** and match what the model expects (after any renames used during training).
- If omitted, the predicted field name is usually `<target_field>_predicted` (varies by MLTK version); use `as` to be explicit.

Examples

```
... | apply m_price_lr as predicted_price

... | apply m_latency_by_svc as yhat
```

C. `summary`

Inspects a saved model (metadata, coefficients, validation).

Shape

```
| summary <model_name>
```

What you'll see

- Algorithm, target, feature list, partitions (if any)
- Coefficients & intercept
- CV or holdback metrics if used in `fit`

Example

```
| summary m_cpu_lr
```

D. `score` (optional for regression)

Computes quality metrics comparing **actual vs. predicted**.

Two ways

1. **Built-in**: `score` (availability/params vary by version)
2. **Manual SPL** (portable & transparent—recommended)

Manual SPL pattern

```
... | eval residual = actual - predicted
    | eval se = pow(residual,2), ae = abs(residual)
    | eventstats avg(se) as MSE avg(ae) as MAE
    | eval RMSE = sqrt(MSE)
    | stats first(RMSE) as RMSE first(MAE) as MAE \
          corr(actual, predicted) as PearsonR
    | eval R2 = pow(PearsonR, 2)
```

---

2) End-to-end Linear Regression patterns

Pattern 1: Clean train → save → apply → score

```
/* TRAIN */
index=infra_metrics sourcetype=perf:host earliest=-7d
| bin _time span=1m
| stats avg(cpu_user) as cpu_user avg(cpu_system) as cpu_system
avg(mem_used) as mem_used by _time host
| eval cpu_load = cpu_user + cpu_system
| fillnull value=0 cpu_user cpu_system mem_used cpu_load
| fit LinearRegression cpu_load from cpu_user cpu_system mem_used
kfold_cv=5 into m_cpu_lr

/* INSPECT */
| summary m_cpu_lr

/* APPLY (new window/search) */
index=infra_metrics sourcetype=perf:host earliest=-15m
| bin _time span=1m
| stats avg(cpu_user) as cpu_user avg(cpu_system) as cpu_system
avg(mem_used) as mem_used by _time host
| eval cpu_load = cpu_user + cpu_system
| fillnull value=0 *
| apply m_cpu_lr as predicted_cpu_load

/* SCORE */
| eval residual = cpu_load - predicted_cpu_load
| eval se=pow(residual,2), ae=abs(residual)
| eventstats avg(se) as MSE avg(ae) as MAE
```

```
| eval RMSE=sqrt(MSE)
| stats first(RMSE) as RMSE first(MAE) as MAE \
        corr(cpu_load, predicted_cpu_load) as PearsonR
| eval R2 = pow(PearsonR,2)
```

Pattern 2: One-model-per-entity (partitioned LR)

```
/* TRAIN one model per service */
index=svc_metrics earliest=-14d
| stats avg(latency_ms) as latency avg(rps) as rps avg(payload_kb) as
payload by _time service
| fillnull value=0 *
| fit LinearRegression latency from rps payload partition_by=service
kfold_cv=5 into m_lat_by_svc

/* APPLY (the right service row uses its own model) */
index=svc_metrics earliest=-1h
| stats avg(latency_ms) as latency avg(rps) as rps avg(payload_kb) as
payload by _time service
| apply m_lat_by_svc as latency_hat
```

---

3) Regularized LR options (when multicollinearity bites)

All share the same command form as `LinearRegression`, just swap the learner and add params.

Ridge (L2)

```
| fit Ridge cpu_load from cpu_user cpu_system mem_used alpha=1.0 into
m_cpu_ridge
```

Lasso (L1)

```
| fit Lasso cpu_load from cpu_user cpu_system mem_used alpha=0.001 into
m_cpu_lasso
```

ElasticNet (L1+L2)

```
| fit ElasticNet cpu_load from cpu_user cpu_system mem_used alpha=0.05
l1_ratio=0.5 into m_cpu_en
```

Tips

- Tune `alpha` (strength) and `l1_ratio` (0=L2, 1=L1).
- Use `kfold_cv` to pick hyperparameters that generalize.

---

4) Feature engineering that plays nicely with `fit`

MLTK doesn't auto-encode everything—prepare features explicitly in SPL:

Numeric scaling (if magnitudes differ a lot)

```
| eventstats avg(x) as mx stdev(x) as sx
| eval x_z = if(sx>0,(x-mx)/sx,0)
```

Robust winsorization (cap extreme outliers)

```
| eval x_capped = case(x<-100,-100, x>100,100, true(),x)
```

One-hot encode categorical

```
| eval os_win=if(os="windows",1,0), os_lin=if(os="linux",1,0),
os_mac=if(os="mac",1,0)
```

Then pass these engineered fields to `fit`.

---

5) Model lifecycle, scope & deployment

- **Model storage**: Saved under the MLTK app's model store (app scope). In **SHC/Cloud**, save in an app shared to the search heads that will `apply` it.
- **Versioning**: Use semantic names: `m_cpu_lr_v1`, `v2_2025_09`, etc.
- **Retraining cadence**: schedule a weekly/monthly job to `fit` and overwrite `into` the same name, or rotate versions and switch dashboards after validation.
- **Dashboards**: Put `apply` + residual charts in a scheduled saved search or base search. Track **RMSE/R²** as KPIs over time.

---

6) Troubleshooting quick hits

- **"Model not found"** → The `apply` search runs in a different app/scope than where you saved it. Re-save `into` in the target app or switch the search's app context/share permissions.
- **"Missing feature(s)"** → Ensure **identical field names** at apply time. Reproduce the **same renames/evals** you used before `fit`.
- **All-zero predictions / NaNs** → Upstream nulls; add `fillnull value=0 *` (or domain-aware fills). Check units/scales.
- **Unstable coefficients (huge magnitudes, flip-flop across folds)** → Features are collinear or on wildly different scales. Drop one of a collinear pair, or try **Ridge/Lasso/ElasticNet**.

- **Poor generalization** → Prefer `kfold_cv` over `holdback` for small datasets; simplify features; inspect residuals and outliers.

---

7) Minimal "cheat sheet"

```
/* Train */
| fit LinearRegression y from x1 x2 x3 kfold_cv=5 into m_lr

/* Inspect */
| summary m_lr

/* Apply */
| apply m_lr as yhat

/* Score (manual) */
| eval resid=y-yhat, se=pow(resid,2), ae=abs(resid)
| eventstats avg(se) as MSE avg(ae) as MAE
| eval RMSE=sqrt(MSE)
| stats first(RMSE) as RMSE first(MAE) as MAE corr(y,yhat) as r
| eval R2=pow(r,2)
```

Splunk DLTK/DSDL – Setup Guide

1) Prerequisites

- **Splunk Enterprise (8.1+)** or **Splunk Cloud Platform** (check your org's allowlist) Splunk Docs
- **Machine Learning Toolkit (MLTK)** app installed on the search head/SHC Splunk Documentation+1
- **Container environment** you control: **Docker** (local or VM) or **Kubernetes/OpenShift** (e.g., EKS) Splunk
- Network route from Splunk (search head) → container endpoint (default ports 5000, 8888)
- (Optional) **GPU** nodes if you plan to use GPU images

Naming: DLTK was renamed to **DSDL** and is Splunk-supported from v5.0+. Splunk

---

2) Install the Apps in Splunk

1. **Install MLTK** (Apps ▸ Find more apps ▸ *Machine Learning Toolkit* ▸ Install). Splunk Documentation
2. **Install DSDL (formerly DLTK)** from Splunkbase (*App ID 4607*). splunkbase.splunk.com

DSDL extends MLTK and ships prebuilt images for TensorFlow, PyTorch, NLP, etc. splunkbase.splunk.com

---

3) Bring Up the DSDL Container(s)

Quick start with Docker (CPU "golden" image)

```
docker run -it --rm --name mltk-container-golden-image-cpu \
  -p 5000:5000 -p 8888:8888 -p 6006:6006 \
  -v mltk-container-data:/srv \
  splunk/mltk-container-golden-image-cpu:5.2.1
```

- 5000 = container endpoint, 8888 = JupyterLab, 6006 = TensorBoard
- Use a GPU-enabled tag if you have GPUs (e.g., `golden-gpu`).
- Official container tags & build scripts live here. GitHub

Kubernetes / OpenShift

- Deploy one (or more) DSDL containers behind a service; map 5000/8888.
- Splunk has a walkthrough with **Amazon EKS** you can follow. Splunk

Air-gapped? Follow the doc to mirror images to your private registry and install dependencies manually. Splunk Documentation

---

4) Wire DSDL to Your Container(s)

In Splunk Web: **Apps ▸ Data Science & Deep Learning**

- Go to **Configuration ▸ Containers**
- Add your container endpoint (e.g., `https://<host>:5000`) and save.
- Ensure the default **`__dev__`** container shows **Running**. Splunk Documentation+1

You'll also create a **Splunk Access Token** in DSDL's config (used by notebooks/containers to call Splunk's REST API). Splunk Documentation

---

5) Permissions & MLTK Integration

- Make sure users who'll run notebooks or jobs have access to MLTK and DSDL app contexts.
- DSDL is designed to work **with** MLTK (it contributes custom algorithms/notebooks). Splunk Docs

---

6) Validate the Setup

- Open DSDL → **JupyterLab** and run a sample notebook (e.g., TensorFlow/PyTorch/NLP).
- Or in the DSDL UI, check **Model Workflow** and verify a dev container is available to run jobs. Splunk Documentation+1

---

7) Splunk Cloud notes

- Docs list **compatibility with Splunk Cloud Platform**, but availability of GPU workers and container connectivity can vary by tenant/security posture—confirm with your Splunk admin/rep. Splunk Docs+1
- Community reports have noted limitations when GPUs aren't provided in Cloud. Treat GPU usage as **environment-dependent**. Reddit

---

8) Security & Hardening tips

- Replace the container's self-signed certs with your own (supported by the container build). GitHub
- Restrict inbound access to 5000/8888, prefer private networking, and rotate Splunk access tokens regularly.
- For enterprise builds, base your own image on Splunk's **UBI** variants and push to your private registry. GitHub

---

9) One-page checklist (TL;DR)

1. Install **MLTK** → Install **DSDL**. Splunk Documentation+1
2. Start a **container** (Docker/K8s) exposing **5000/8888**. GitHub
3. In DSDL: **Configuration** ▸ **Containers** → add endpoint; confirm **`__dev__` running**. Splunk Documentation

4. Create **Splunk Access Token** in DSDL config. Splunk Documentation
5. Open **JupyterLab** in DSDL → run a sample notebook. Splunk Documentati