Author: Dr. Mallarapu
Created: 2026-07-27
Course: SEAS 8414 — Security Analytics

Goal of this notebook

Train and audit detectors on CIC-ToN-IoT network flows re-featured by CICFlowMeter.

What you will learn

1. Read a majority-class baseline before trusting any accuracy figure.
2. Find the strongest single feature, then test it by dropping it and refitting.
3. Tell duplicate inflation apart from genuine signal.
4. Report per-group recall, because the rare classes carry the risk.

Where this connects to the course text

The text builds a defence pipeline; this notebook trains a classifier and audits it. The links below are to specific chapter objectives that share an *analytic move*, not to matching subject matter.

- **Chapter 2: Service Enrichment and Device Fingerprinting** — Learning objective 3 (section 2.1) compares evidence sources by strength and **spoofability**. The features here are CICFlowMeter flow statistics - durations, inter-arrival times, packet-size aggregates - not protocol fields. An attacker moves them indirectly, by pacing and padding traffic. That is a weaker form of the same spoofability concern.
- **Chapter 9: Supply-Chain Integrity and Counterfeit Detection** — Learning objectives 2 and 3 (section 9.1) separate documentation gaps from tamper evidence, and warn that screening signals are **not independent**. Both apply to artifact-derived features.
- **Chapter 11: Formal Protocol Verification** — Section **11.1.2**, titled *Proved, tested, and hoped*, asks you to separate exactly those three. (Chapter 11 lists its objectives in §11.0, not §11.1 as the other chapters do.) The ablation does that job here: it tests whether the headline survives.

IoT/IIoT Network-Flow Intrusion Detection: CIC-ToN-IoT

Model comparison + validity audit on CIC-ToN-IoT **network flows** (≥1M records, via Kaggle)

Abstract: CIC-ToN-IoT has **4,847,499** labeled flow records. It is a CICFlowMeter reprocessing of the UNSW ToN_IoT **network-traffic** captures — the pcap side of ToN_IoT, not its sensor-telemetry tables (Moustafa, 2021). **Every feature used below is a CICFlowMeter flow statistic.** None of ToN_IoT's seven sensor-telemetry tables (fridge, GPS tracker, motion light, garage door, Modbus, thermostat, weather) and none of its OS/audit logs are read here. So nothing in this notebook is evidence about *telemetry-based* detection. We compare four learners and audit the near-balanced, near-perfect scores.

1. Research problem

Task: Classify IoT/IIoT **network flows** as benign or attack. The families that reach the per-family recall breakdown in §9 include `mitm`, `ddos`, `dos`, `ransomware`, `backdoor` and `scanning`.

Scope, stated once so it is not assumed away: ToN_IoT is a multi-view corpus — sensor telemetry, OS/audit logs, and raw network captures from an IIoT testbed. This notebook reads **only** the CICFlowMeter re-featurisation of the network captures. Telemetry is named here to mark what we are *not* using: a flow-only detector says nothing about whether device telemetry would have caught the same attacks.

2. Literature review

- **Moustafa (2021)** — ToN_IoT: a new generation of IoT/IIoT datasets for the whole telemetry, OS-log and network stack.
- **Booij, Chiscop, Meeuwissen, Moustafa & den Hartog (2022)** — *ToN_IoT: The Role of Heterogeneity and the Need for Standardization of Features and Attack Types in IoT Network Intrusion Data Sets* (IEEE Internet of Things Journal).
- **Sommer & Paxson (2010)** — the closed-world critique.

Related approaches and their known caveats — drawn from the wider literature; these are **not** measurements reproduced on this exact corpus:

Reported approach	Known caveat
Moustafa (2021) — ML baselines	in-distribution testbed
Various IoT-NIDS papers	flow features from one testbed

3. Dataset provenance & honesty caveats

Property	Value
Source	Kaggle `dhoogla/cictoniot` (CIC-ToN-IoT-V2)

Property	Value
View	network flows only — no telemetry table, no OS/audit log
Rows	4,847,499 flow records
Features	68 numeric flow statistics after constant-column removal (printed below)
Label	`Label` 0/1; family `Attack`
Access	Kaggle API token required

Honestly: `Attack` (the family name) is dropped from features to prevent leakage; one testbed, so cross-distribution generalization is untested.

Preprocessing caveat that §10 and §11 depend on — read it before quoting any duplicate rate. The loader ends with `X.clip(-1e15, 1e15)`, a blunt guard against float overflow, and then deletes any column left with a single distinct value. That clip is not free. Every value beyond the bound is mapped onto the *same* number. So (i) two flows that differed only above the bound become byte-identical rows. And (ii) a column lying entirely above the bound becomes constant and is dropped. In this corpus the clip does bite. The CICFlowMeter `Idle Mean` / `Idle Std` / `Idle Max` / `Idle Min` fields are recorded on a scale that runs past the 1e15 bound. That is a property of the source parquet, not something a cell below prints. `Idle Max` is the one column the clip alone removes. Consequence: the duplicate and overlap rates printed in §10 are measured on this post-clip matrix, and part of what they count is duplication *we* manufactured.

Before you run this: getting the data

This notebook downloads its own data on the first run, then caches it. You do not fetch anything by hand.

Dataset: Kaggle `dhoogla/cictoniot` -> `/tmp/kg_cictoniot`. It is about **420 MB** on disk.

One-time setup. Sign in at kaggle.com, open **Settings**, and under **API** choose **Create New Token**. Kaggle hands you a `kaggle.json` file. This notebook does *not* read that file. It reads a plain key file, so convert it once:

```
mkdir -p ~/.kaggle
python3 -c "import json;print(json.load(open('kaggle.json'))
['key'],end='')" > ~/.kaggle/access_token
chmod 600 ~/.kaggle/access_token
```

Never paste the token into a cell, a commit, or a screenshot. If it leaks, revoke it from the same Settings page.

If the loader fails:

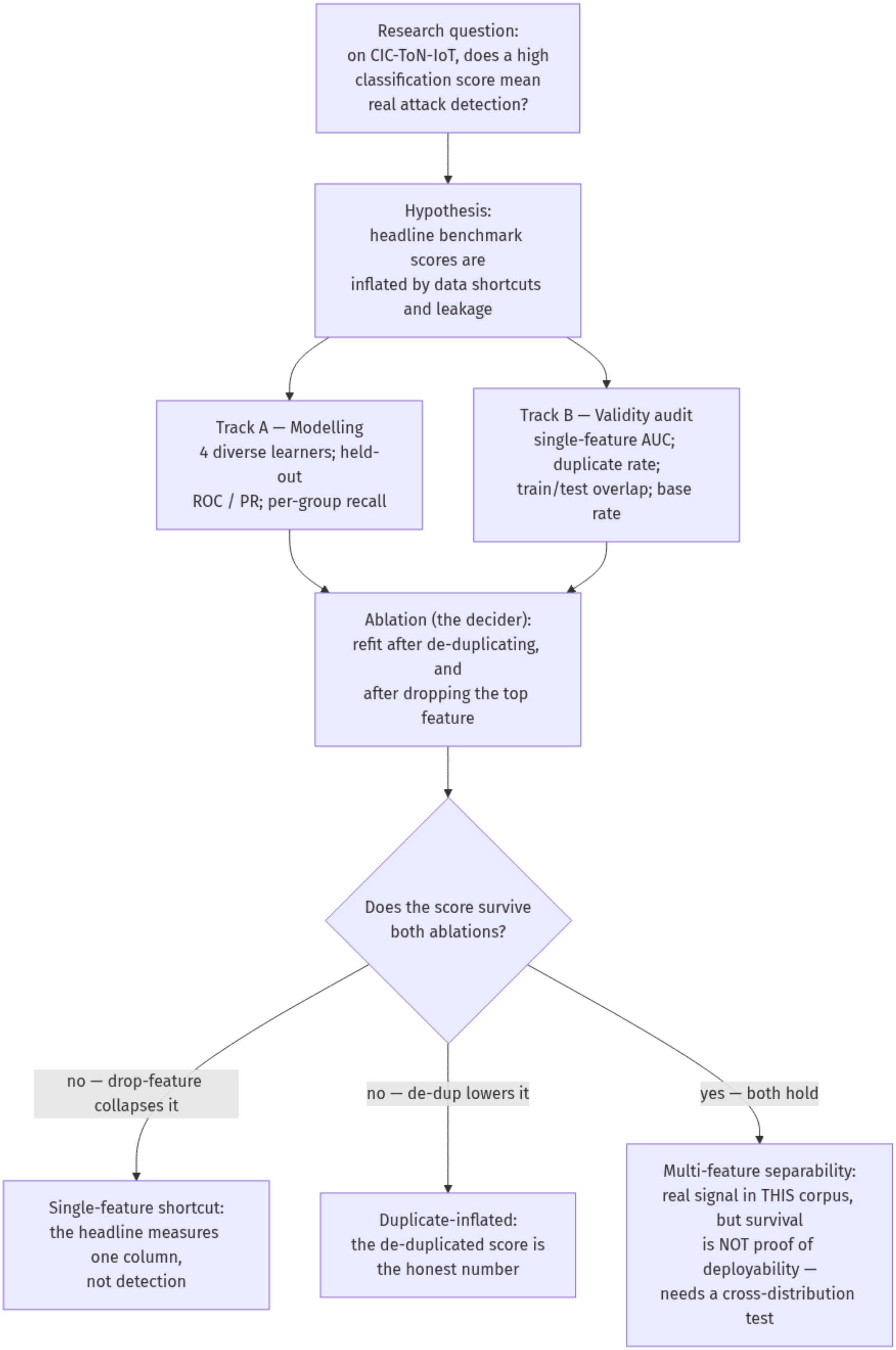
- `FileNotFoundError: ~/.kaggle/access_token` - you created `kaggle.json` but not the key file. Run the command above.
- `401 Unauthorized` - the key is wrong, or a trailing newline crept in.
- `403 Forbidden` - open the dataset page on Kaggle while signed in, accept its terms, then re-run the cell.

The cache sits under `/tmp`, which macOS clears on reboot. To keep it, move the folder somewhere durable and symlink it back. Do **not** edit the path in the code cell below: that changes a code cell and invalidates the stored outputs you are reviewing.

4. Solution design

The methodology is deliberately two-track. We *earn* a headline score with standard modelling, then *interrogate* it with a validity audit. Only a verdict that survives both is reported. The diagram below is the shape of every notebook in this series.

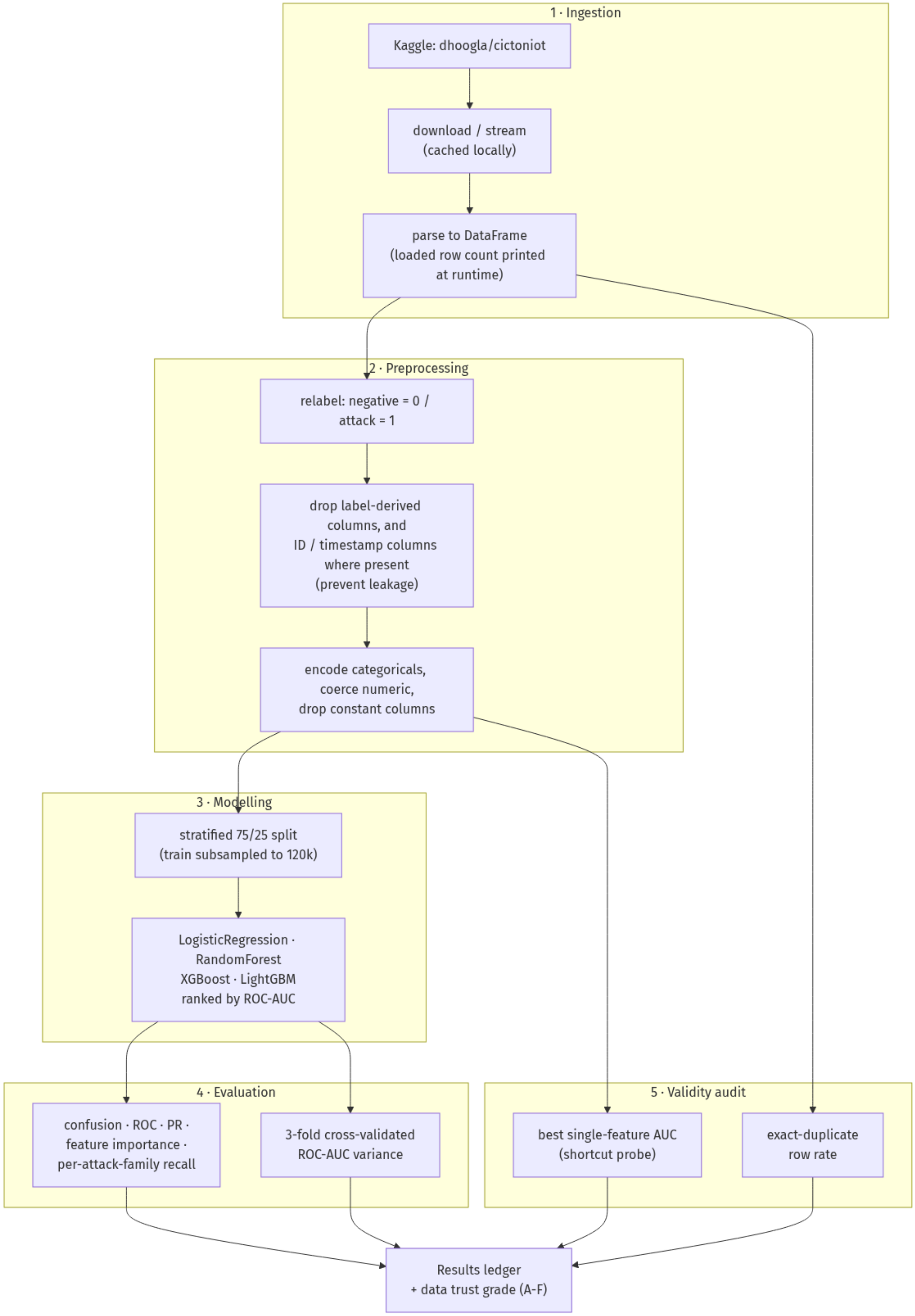
Figure 4.1 — Solution design (methodology).



5. Implementation architecture

Five stages — ingestion, preprocessing, modelling, evaluation, and a parallel validity-audit path — feed a single graded results ledger. Leakage defences (dropping label-derived and identifier columns) live in preprocessing, before any model sees the data.

Figure 5.1 — Implementation architecture.



6. Data acquisition & preparation

Every line below is commented so a student can re-run and modify each step. The cell ends by producing the standard analysis variables: `df`, `X` (clean numeric features), `y` (binary label), `feat` (feature names), and `family`. `family` is the per-group label used for the recall breakdown. It is an attack family on the intrusion corpora, but a transaction type, merchant category or malware category on the fraud/malware ones.

In [1]:
```
%matplotlib inline
import time, warnings; warnings.filterwarnings('ignore')   # keep output
clean
import numpy as np, pandas as pd                             # numerics +
dataframes
import matplotlib.pyplot as plt                              # static plots
(embed in HTML+PDF)
plt.rcParams['figure.dpi'] = 120                             # crisp figures
RANDOM_STATE = 0                                             # single seed
used everywhere
np.random.seed(RANDOM_STATE)                                 # reproducible
sampling
NEG_WORD, POS_WORD = 'benign', 'attack'                       # class names
(overridden by some loaders)
```

In [2]:
```
import os, glob
# Kaggle auth: token read from ~/.kaggle/access_token (students supply
their own).
os.environ.setdefault('KAGGLE_KEY',
open(os.path.expanduser('~/.kaggle/access_token')).read().strip())
import kaggle; kaggle.api.authenticate()
REF = 'dhoogla/cictoniot'; DEST = '/tmp/kg_' + REF.split('/')[-1]
if not os.path.exists(DEST):                                    # download +
unzip once (cached)
    kaggle.api.dataset_download_files(REF, path=DEST, unzip=True,
quiet=True)
NROWS = 1_500_000                                               # per-file
read cap (memory bound)
files = sorted(glob.glob(DEST + '/**/*.parquet', recursive=True))
df = pd.concat([pd.read_parquet(f) for f in files], ignore_index=True)  #
combine day/part files
df.columns = [str(c).strip() for c in df.columns]             # strip
header whitespace
LABEL = 'Label'; FAMILY = 'Attack'
df['y'] = (df[LABEL].astype(str).str.strip().str.lower() !=
'0').astype(int)  # benign=0
df['family'] = df[FAMILY].astype(str).str.strip()             # descriptive
attack family
df = df.reset_index(drop=True)
assert len(df) >= 1_000_000, f'floor not met: {len(df):,}'    # honesty
gate: >= 1M rows
DROP = list({LABEL, FAMILY, 'y', 'family'} | set([]))  # never leak label
cols
feat = [c for c in df.columns if c not in DROP]
from sklearn.preprocessing import LabelEncoder
X = df[feat].copy()
idlike = [c for c in X.select_dtypes(include='object').columns if
X[c].nunique() > 0.5*len(X)]
```

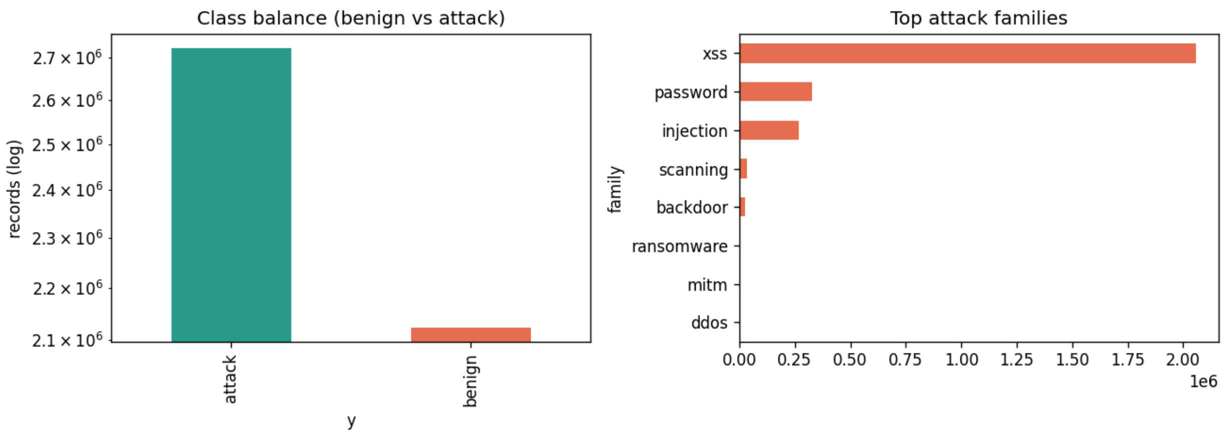
```
X = X.drop(columns=idlike)                                    # drop
ID/timestamp-like leaky columns
for c in X.select_dtypes(include='object').columns:          # encode
remaining categoricals
    X[c] = LabelEncoder().fit_transform(X[c].astype(str))
X = X.apply(pd.to_numeric, errors='coerce').replace([np.inf,-
np.inf],np.nan).fillna(0.0)
X = X.clip(-1e15, 1e15)                                       # clip huge
NetFlow counts (float32-safe)
X = X.loc[:, X.nunique() > 1]                                 # drop
constants
import re                                                     # LightGBM
rejects special chars in names
_seen, _cols = {}, []
for _c in X.columns:                                          # sanitize
to unique, safe names
    _c = re.sub(r'[^0-9A-Za-z_]+', '_', str(_c)).strip('_') or 'f'
    _seen[_c] = _seen.get(_c, -1) + 1
    _cols.append(_c if _seen[_c] == 0 else f'{_c}_{_seen[_c]}')
X.columns = _cols; feat = list(X.columns)
y = df['y'].to_numpy()                                        # STANDARD
CONTRACT
NEG_WORD, POS_WORD = 'benign', 'attack'              # class names for
plots
print(f'loaded {len(df):,} rows x {len(feat)} features; positive rate
{y.mean():.4f}')
```

```
loaded 4,847,499 rows x 68 features; positive rate 0.5619
```

7. Exploratory data analysis

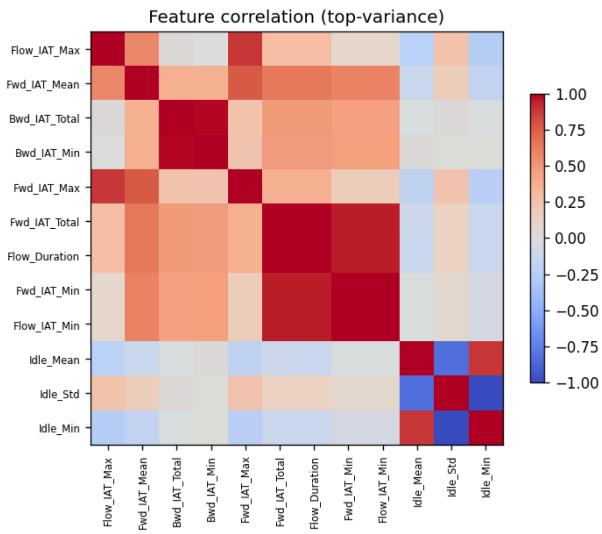
In [3]:

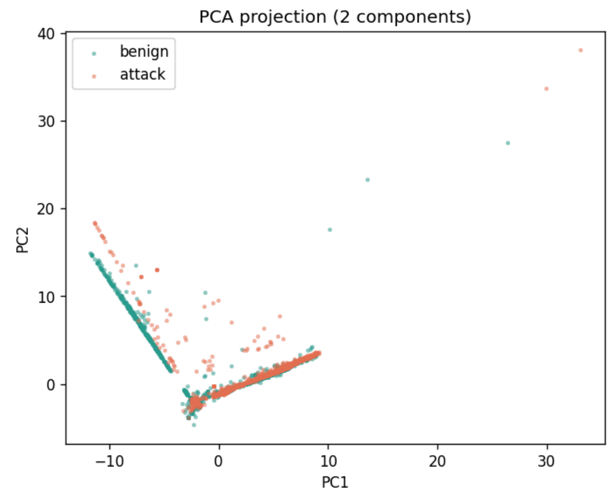
```
# --- EDA 1: class balance and the attack-family mix ---
fig, ax = plt.subplots(1, 2, figsize=(11, 4))
df['y'].map({0:NEG_WORD,1:POS_WORD}).value_counts().plot.bar(
# counts per class
    ax=ax[0], color=['#2a9d8f','#e76f51']); ax[0].set_yscale('log')
ax[0].set_title(f'Class balance ({NEG_WORD} vs {POS_WORD})');
ax[0].set_ylabel('records (log)')
df.loc[df.y==1,'family'].value_counts().head(8).plot.barh(
# top attack families
    ax=ax[1], color='#e76f51'); ax[1].invert_yaxis(); ax[1].set_title('Top
attack families')
plt.tight_layout(); plt.show()
```



In [4]:

```
# --- EDA 2: feature correlation + a 2-D PCA projection ---
from sklearn.preprocessing import StandardScaler                # scale
before PCA
from sklearn.decomposition import PCA
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
topv = X[feat].var().sort_values().tail(12).index               # 12
highest-variance features
im = ax[0].imshow(X[topv].corr(), cmap='coolwarm', vmin=-1, vmax=1)  #
correlation heatmap
ax[0].set_xticks(range(len(topv))); ax[0].set_xticklabels(topv,
rotation=90, fontsize=7)
ax[0].set_yticks(range(len(topv))); ax[0].set_yticklabels(topv, fontsize=7)
ax[0].set_title('Feature correlation (top-variance)'); fig.colorbar(im,
ax=ax[0], shrink=0.7)
samp = X.sample(min(5000, len(X)), random_state=RANDOM_STATE)     #
subsample for a fast PCA
pc =
PCA(n_components=2).fit_transform(StandardScaler().fit_transform(samp))
ys = y[samp.index]                                                # aligned
labels for coloring
for lab,c in [(0,'#2a9d8f'),(1,'#e76f51')]:
    ax[1].scatter(pc[ys==lab,0], pc[ys==lab,1], s=4, alpha=0.4, color=c,
                  label={0:NEG_WORD,1:POS_WORD}[lab])
ax[1].set_title('PCA projection (2 components)'); ax[1].legend();
ax[1].set_xlabel('PC1'); ax[1].set_ylabel('PC2')
plt.tight_layout(); plt.show()
```





8. Model comparison

Four diverse learners share one held-out split, ranked by ROC-AUC.

Two honesty guards print with the table:

1. The models train on a *stratified subsample* of at most 120,000 rows. The full row count is printed above. So every score here is a subsample number, not a full-corpus claim.
2. The **majority-class baseline accuracy** appears *inside* the ranking table. On imbalanced data, 0.99 accuracy can be worse than always guessing the majority class. Judge each model against that baseline, not against 0.5.

In [5]:
```
# --- Model comparison: four learners on the same held-out split ---
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, roc_auc_score
import xgboost as xgb, lightgbm as lgb

# Stratified split keeps the class ratio in both halves.
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.25,
random_state=RANDOM_STATE, stratify=y)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
N_MATERIALIZED = len(y)                                    # the full
corpus we loaded (see printed count)
# HONEST DISCLOSURE: we do NOT train on all N. We fit on a STRATIFIED
subsample (<=120k) because
# these learners saturate long before then on this data. Every headline
below is a SUBSAMPLE
# number, not a full-corpus number — saying otherwise would be the
fabrication this course forbids.
if len(Xtr) > 120_000:
    Xtr, _, ytr, _ = train_test_split(Xtr, ytr, train_size=120_000,
random_state=RANDOM_STATE,
```

```
                                        stratify=ytr)                # 
genuinely stratified, not random
MAJORITY_BASELINE = max(np.mean(yte), 1 - np.mean(yte))          # accuracy 
of 'always predict majority'
print(f'materialized {N_MATERIALIZED:,} rows | trained on {len(Xtr):,} 
(stratified subsample) | '
      f'held-out {len(yte):,}')
print(f'MAJORITY-CLASS BASELINE accuracy = {MAJORITY_BASELINE:.4f}  '
      f'(any model must beat THIS, not 0.5, to be interesting)')

models = {                                                          # four 
standard, diverse learners
    'LogisticRegression': make_pipeline(StandardScaler(), 
LogisticRegression(max_iter=300)),  # scaled!
    'RandomForest': RandomForestClassifier(n_estimators=60, n_jobs=-1, 
random_state=RANDOM_STATE),
    'XGBoost': xgb.XGBClassifier(n_estimators=80, max_depth=6, 
tree_method='hist', n_jobs=-1,
                                 eval_metric='logloss', 
random_state=RANDOM_STATE),
    'LightGBM': lgb.LGBMClassifier(n_estimators=80, n_jobs=-1, verbose=-1, 
random_state=RANDOM_STATE),
}
rows, fitted = [], {}
for name, m in models.items():                                      # fit + 
score each model
    t = time.perf_counter(); m.fit(Xtr, ytr); fitted[name] = m
    p = m.predict_proba(Xte)[:, 1]                                  # 
positive-class probability on held-out
    rows.append({'model': name, 'accuracy': round(accuracy_score(yte, 
(p>0.5).astype(int)), 6),
                 'roc_auc': round(roc_auc_score(yte, p), 6),      # 6 dp: a 
1.000000 is a red flag, not a win
                 'train_s': round(time.perf_counter()-t, 1)})
rows.append({'model': 'MajorityBaseline', 'accuracy': 
round(MAJORITY_BASELINE, 4),
             'roc_auc': 0.5, 'train_s': 0.0})             # show the 
baseline IN the ranking table
comparison = pd.DataFrame(rows).sort_values('roc_auc', 
ascending=False).reset_index(drop=True)
_ranked = comparison[comparison.model != 'MajorityBaseline']
best_name = _ranked.iloc[0]['model']; best = fitted[best_name]  # winner by 
ROC-AUC (excl. baseline)
print('best model:', best_name); comparison
```

```
materialized 4,847,499 rows | trained on 120,000 (stratified subsample) | 
held-out 1,211,875
MAJORITY-CLASS BASELINE accuracy = 0.5619  (any model must beat THIS, not 
0.5, to be interesting)
best model: LightGBM
```

Out [5]:

	model	accuracy	roc_auc	train_s
0	LightGBM	0.991056	0.996751	1.5
1	XGBoost	0.991390	0.996673	0.9
2	RandomForest	0.990370	0.993853	2.5
3	LogisticRegression	0.901802	0.940597	1.3
4	MajorityBaseline	0.561900	0.500000	0.0

9. Results

Diagnostics for the winning model, including **per-group recall**.

The grouping comes from whatever the loader put in `family`. It is *not* always an attack taxonomy. On the intrusion corpora it is the attack family. On the fraud and malware corpora it is a transaction type, a merchant category or a malware category. On binary corpora it collapses to the positive class.

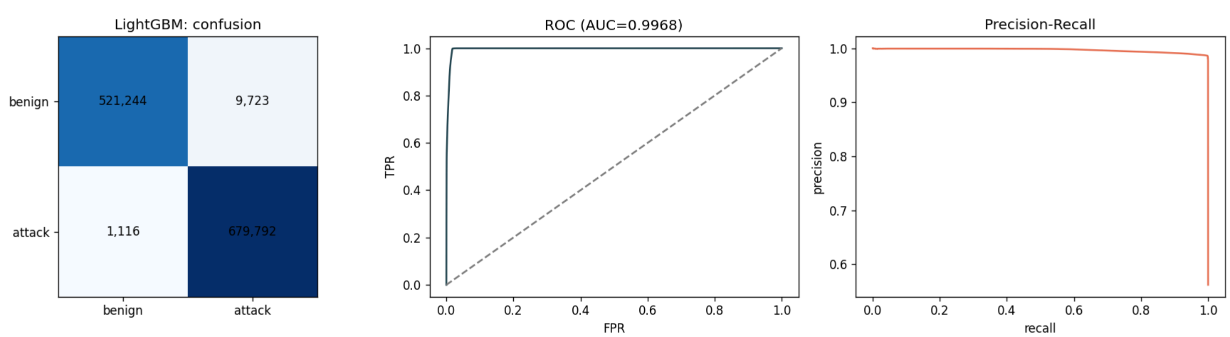
Read it accordingly. Where the groups are genuinely rare classes, they reveal whether detection is real. The dominant flood classes do not.

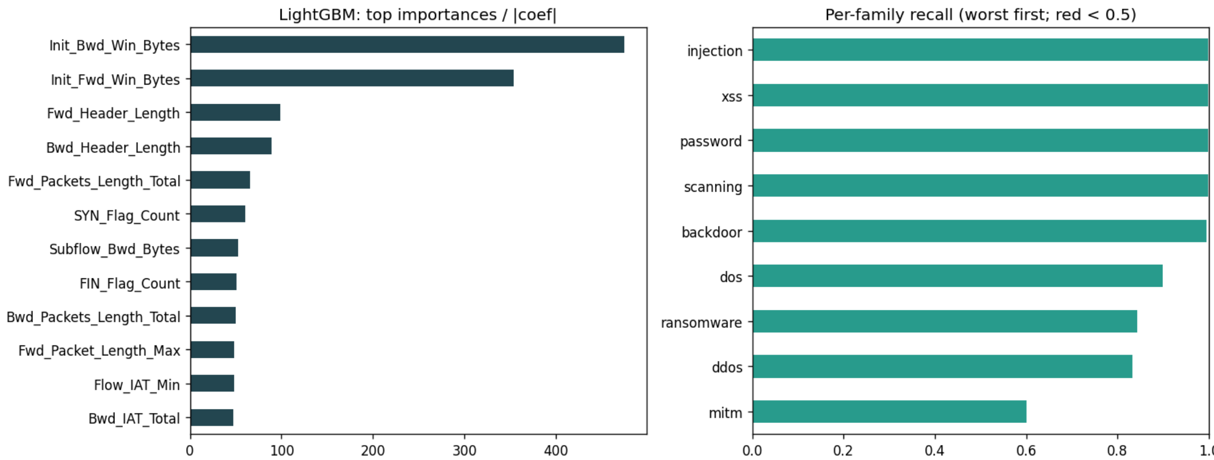
In [6]:

```
# --- Results for the best model: confusion, ROC, PR, importances, per-
family recall ---
from sklearn.metrics import confusion_matrix, roc_curve,
precision_recall_curve, recall_score
pb = best.predict_proba(Xte)[:, 1]; pred = (pb > 0.5).astype(int)
fig, ax = plt.subplots(1, 3, figsize=(15, 4))
# (1) confusion matrix
cm = confusion_matrix(yte, pred); ax[0].imshow(cm, cmap='Blues')
ax[0].set_title(f'{best_name}: confusion'); ax[0].set_xticks([0,1]);
ax[0].set_yticks([0,1])
ax[0].set_xticklabels([NEG_WORD,POS_WORD]);
ax[0].set_yticklabels([NEG_WORD,POS_WORD])
for (i,j),v in np.ndenumerate(cm):
ax[0].text(j,i,f'{v:,}',ha='center',va='center')
# (2) ROC and PR curves
fpr,tpr,_ = roc_curve(yte, pb); prec,rec,_ = precision_recall_curve(yte,
pb)
ax[1].plot(fpr,tpr,color='#264653'); ax[1].plot([0,1],[0,1],'--',c='grey')
ax[1].set_title(f'ROC (AUC={roc_auc_score(yte,pb):.4f})');
ax[1].set_xlabel('FPR'); ax[1].set_ylabel('TPR')
ax[2].plot(rec,prec,color='#e76f51'); ax[2].set_title('Precision-Recall');
ax[2].set_xlabel('recall'); ax[2].set_ylabel('precision')
plt.tight_layout(); plt.show()

# (3) feature importances + (4) per-attack-family recall
fig, ax = plt.subplots(1, 2, figsize=(13, 5))
imp, names = None, feat                                       #
importances, robust to the scaled-LR pipeline
if hasattr(best, 'feature_importances_'):                      # tree
```

```
models
    imp = best.feature_importances_; names = list(getattr(best,
'feature_names_in_', feat))[:len(imp)]
elif hasattr(best, 'named_steps') and 'logisticregression' in getattr(best,
'named_steps', {}):
    imp = np.abs(best.named_steps['logisticregression'].coef_[0]); names =
feat  # LR pipeline
elif hasattr(best, 'coef_'):
    imp = np.abs(best.coef_[0]); names = feat
if imp is not None:
    pd.Series(imp,
index=names[:len(imp)]).sort_values().tail(12).plot.barh(ax=ax[0],
color='#264653')
ax[0].set_title(f'{best_name}: top importances / |coef|')
# Per-family recall, WORST-first so rare, hard classes are visible, not
just the dominant floods.
fam_te = df.loc[Xte.index, 'family']
fr = {}
for fam, cnt in fam_te[yte==1].value_counts().items():
    if cnt < 5: continue                                          # need a
few positives for a meaningful recall
    mask = (fam_te==fam).to_numpy(); fr[fam] = recall_score(yte[mask],
pred[mask], zero_division=0)
srt = pd.Series(fr).sort_values()
show = pd.concat([srt.head(9), srt.tail(3)]) if len(srt) > 12 else srt   #
worst 9 + best 3
show = show[~show.index.duplicated()]
show.plot.barh(ax=ax[1], color=['#e76f51' if v < 0.5 else '#2a9d8f' for v
in show]); ax[1].set_xlim(0,1)
ax[1].set_title('Per-family recall (worst first; red < 0.5)')
plt.tight_layout(); plt.show()
# Operational numbers, not just figures: false-positive rate and the worst
per-family recalls.
tn, fp = int(cm[0,0]), int(cm[0,1])
fpr_op = fp/(fp+tn) if (fp+tn) > 0 else float('nan')          # benign
wrongly flagged @0.5
print(f'operational FALSE-POSITIVE RATE @0.5 = {fpr_op:.4f}  ({fp:,} benign
flagged of {fp+tn:,})')
print('worst per-family recalls:', {k: round(v, 3) for k, v in
srt.head(6).items()})
```






```
operational FALSE-POSITIVE RATE @0.5 = 0.0183  (9,723 benign flagged of
530,967)
worst per-family recalls: {'mitm': 0.602, 'ddos': 0.833, 'ransomware': 0.844,
'dos': 0.9, 'backdoor': 0.995, 'scanning': 0.998}
```

10. Validity audit — is the score real?

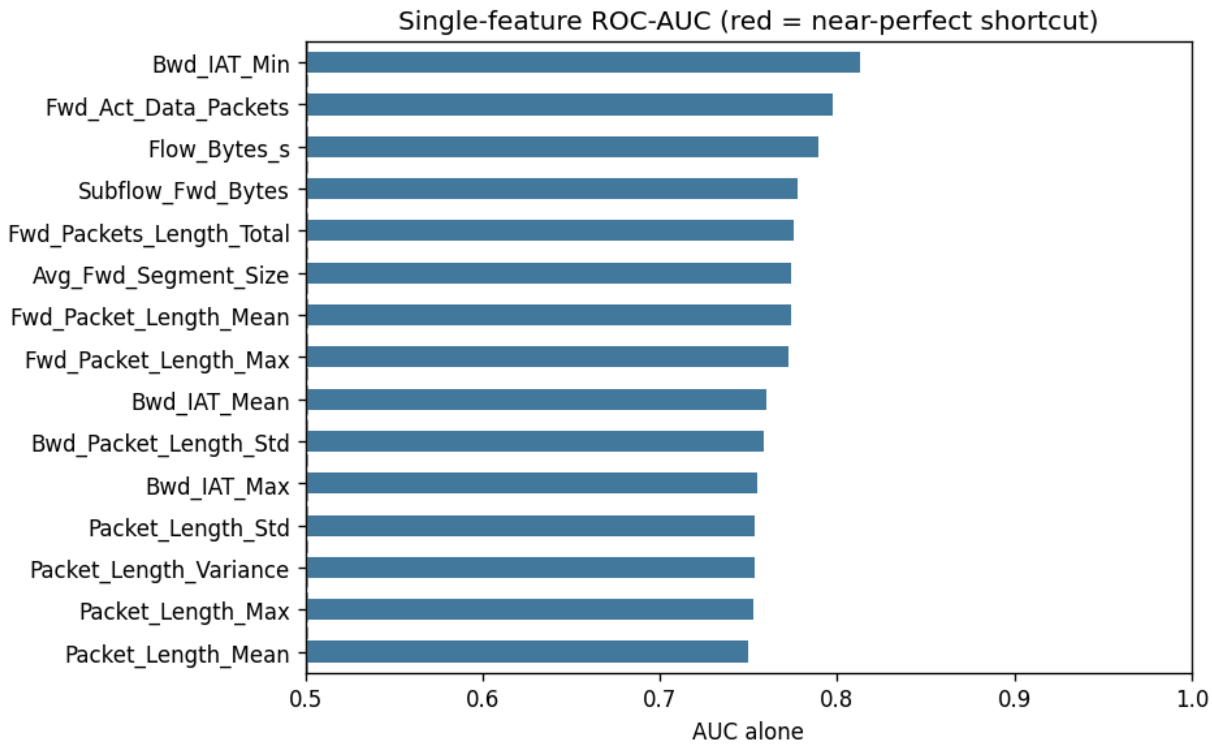
Three diagnostics. **(a)** How well can the *single best feature*, alone, separate the classes? A near-1.0 single-feature AUC means that feature is *near-sufficient* — a shortcut (which may be legitimate signal or an artifact), not the same as target leakage. **(b)** The exact-duplicate row rate. **(c)** The **train/test exact-row contamination** — the fraction of held-out rows that are duplicates of training rows, which is what actually inflates a held-out score. The trust grade is the *worse* of the single-feature and contamination concerns.

What these numbers are measured on — read this before quoting them: All three diagnostics run on `X`, the matrix *after* preprocessing, never on the raw parquet. Because of the ±1e15 clip described in §3, rows that differed only in the out-of-range `Idle` fields arrive at this cell already identical. So **(b)** and **(c)** are **upper bounds** on duplication and split leakage in the source capture. Some part of each is an artifact of our own pipeline, not of CIC-ToN-IoT. This notebook does not measure the pre-clip rates. So it quotes no split between manufactured and native duplication. Reading the printed value as a property of the dataset would be wrong. The same clip constrains **(a)** in two ways. A column the clip drives to a single value is dropped before this cell runs, and cannot be ranked at all. A column it only partially flattens *is* still ranked — but on degraded values. So “no single feature dominates” is a claim about the post-clip feature set, not about everything CICFlowMeter measured.

In [7]:
```
# --- Validity audit: is the score real detection, or a data shortcut? ---
from sklearn.metrics import roc_auc_score
samp = X.sample(min(60_000, len(X)), random_state=1); ysamp = y[samp.index]
aucs = {}
for c in feat:                                                    # AUC of
EACH feature alone
    col = samp[c].to_numpy(float)
    if col.std()==0: continue
    a = roc_auc_score(ysamp, col); aucs[c] = max(a, 1-a)          #
```

```python
direction-agnostic
best_auc = max(aucs.values()); best_col = max(aucs, key=aucs.get)
dup_rate = 1 - X.drop_duplicates().shape[0]/len(X)                # exact-
duplicate feature rows (whole set)
# The statistic that actually inflates a held-out score is TRAIN/TEST
CONTAMINATION: how many test
# rows are exact duplicates of a training row. Measure it directly on the
split used above.
_trkeys = set(map(tuple, np.round(Xtr.to_numpy(), 6)))
_te = np.round(Xte.to_numpy(), 6)[:50_000]
contam = float(np.mean([tuple(r) in _trkeys for r in _te]))      # fraction
of test rows seen in train
# Trust grade reflects BOTH failure modes and takes the WORSE of the two: a
near-perfect single
# feature (shortcut) OR heavy train/test contamination each independently
invalidate the headline.
_ga = 'F' if best_auc>=0.999 else 'D' if best_auc>=0.99 else 'C' if
best_auc>=0.95 else 'B' if best_auc>=0.85 else 'A'
_gc = 'F' if contam>=0.5 else 'D' if contam>=0.3 else 'C' if contam>=0.15
else 'B' if contam>=0.05 else 'A'
grade = max(_ga, _gc)                                              # 'max'
letter = worse grade (A best, F worst)
print(f'best single-feature AUC = {best_auc:.4f}  (feature: {best_col})')
print(f'   note: a near-1.0 single-feature AUC means this feature is *near-
sufficient* (a shortcut),\n'
      f'   which may be legitimate signal OR an artifact - it is NOT the
same as target leakage.')
print(f'exact-duplicate row rate (whole corpus) = {dup_rate:.3f}')
print(f'TRAIN/TEST exact-row contamination       = {contam:.3f}  (single-
feat grade {_ga}, contam grade {_gc})')
print(f'==> data trust grade: {grade}   (worse of the two; F = shortcut
and/or heavy contamination)')
s = pd.Series(aucs).sort_values().tail(15)
fig, ax = plt.subplots(figsize=(8,5))
s.plot.barh(ax=ax, color=['#e76f51' if v>=0.99 else '#457b9d' for v in s]);
ax.axvline(0.5,ls='--',c='grey')
ax.set_xlim(0.5,1.0); ax.set_title('Single-feature ROC-AUC (red = near-
perfect shortcut)'); ax.set_xlabel('AUC alone')
plt.tight_layout(); plt.show()
```

```
best single-feature AUC = 0.8132  (feature: Bwd_IAT_Min)
   note: a near-1.0 single-feature AUC means this feature is *near-
sufficient* (a shortcut),
   which may be legitimate signal OR an artifact - it is NOT the same as
target leakage.
exact-duplicate row rate (whole corpus) = 0.307
TRAIN/TEST exact-row contamination       = 0.114  (single-feat grade A,
contam grade B)
==> data trust grade: B   (worse of the two; F = shortcut and/or heavy
contamination)
```



11. Ablation — does the headline survive removing the artifacts?

Narrating a shortcut is not enough. We *retrain the winning model* after (1) de-duplicating the corpus (removing the train/test contamination) and (2) dropping the single strongest feature. We report the held-out AUC each time. **Read the result honestly, both ways:** if the AUC **collapses**, the headline was a contamination/shortcut artifact. If it **barely moves** — common on *simulated* corpora — that is **not vindication**. It means the classes are separable by *many* redundant features, because the attack and benign distributions barely overlap. That is its own generation artifact. The numbers below decide which story is true here, not the prose.

Caveat carried down from §3: the de-duplication variant drops every row that is a duplicate *after* clipping, which includes rows the clip itself made identical. It therefore removes more than the capture's own duplication, and the “rows removed” percentage in the table should be read as an upper bound too. That makes the de-duplicated AUC a *conservative* check — it strips out more data than a clean pipeline would.

In [8]:
```
# --- Ablation: SHOW the inflation empirically, don't just narrate it ---
from sklearn.base import clone
def _retrain_auc(Xa, ya):                                              # re-
split, stratified-subsample, refit best family
    xtr, xte, ytr2, yte2 = train_test_split(Xa, ya, test_size=0.25,
random_state=RANDOM_STATE, stratify=ya)
    if len(xtr) > 120_000:
        xtr, _, ytr2, _ = train_test_split(xtr, ytr2, train_size=120_000,
random_state=RANDOM_STATE, stratify=ytr2)
```

```
    m = clone(best); m.fit(xtr, ytr2)
    return roc_auc_score(yte2, m.predict_proba(xte)[:, 1])
base_auc = roc_auc_score(yte, best.predict_proba(Xte)[:, 1])     # (0) the
headline held-out AUC
Xdd = X.drop_duplicates(); ydd = y[Xdd.index]                    # (1) de-
duplicated corpus
auc_dedup = _retrain_auc(Xdd, ydd)
auc_noshort = _retrain_auc(X.drop(columns=[best_col]), y) if best_col in
X.columns else base_auc  # (2) drop shortcut
ablation = pd.DataFrame([
    {'setting': 'headline (as-is)',             'held_out_auc':
round(base_auc, 6)},
    {'setting': f'de-duplicated ({1-len(Xdd)/len(X):.0%} rows removed)',
'held_out_auc': round(auc_dedup, 6)},
    {'setting': f'shortcut feature dropped ({best_col})', 'held_out_auc':
round(auc_noshort, 6)},
])
print('Ablation — how much of the headline survives once each artifact is
removed:')
ablation
```

```
Ablation — how much of the headline survives once each artifact is removed:
```

Out[8]:

	setting	held_out_auc
0	headline (as-is)	0.996751
1	de-duplicated (31% rows removed)	0.995080
2	shortcut feature dropped (Bwd_IAT_Min)	0.996740

12. Reproducibility & robustness

In [9]:

```
# --- Reproducibility & robustness ---
import sklearn
from sklearn.model_selection import StratifiedKFold, cross_val_score
print(f'seed={RANDOM_STATE} | numpy {np.__version__} | sklearn
{sklearn.__version__} | '
      f'xgboost {xgb.__version__} | lightgbm {lgb.__version__}')
# 3-fold cross-validated ROC-AUC of the winning model (fresh clone, bounded
subsample) -> mean +/- std.
from sklearn.base import clone
cvX, cvy = Xtr.iloc[:40_000], ytr[:40_000]
def _auc_scorer(est, Xv, yv):                                  # robust to
xgboost's 2-col predict_proba
    p = est.predict_proba(Xv)
    p = p[:, 1] if getattr(p, 'ndim', 1) == 2 else p
    return roc_auc_score(yv, p)
try:
    cv = cross_val_score(clone(best), cvX, cvy,
                         cv=StratifiedKFold(3, shuffle=True,
random_state=RANDOM_STATE),
                         scoring=_auc_scorer, error_score='raise')
    assert np.all(np.isfinite(cv)), 'non-finite CV folds'   # FAIL CLOSED:
never narrate a NaN as evidence
    print(f'{best_name} 3-fold CV ROC-AUC = {cv.mean():.4f} +/-
```

```
{cv.std():.4f}  '
          f'(mean +/- std across 3 stratified folds; a small std means a
stable estimate on this split)')
except Exception as e:
    print(f'CV UNAVAILABLE ({type(e).__name__}: {str(e)[:60]}); rely on the
single held-out AUC above - '
          f'we do NOT report a CV number we could not compute')
```

```
seed=0 | numpy 2.3.5 | sklearn 1.9.0 | xgboost 1.6.2 | lightgbm 4.7.0
LightGBM 3-fold CV ROC-AUC = 0.9962 +/- 0.0002  (mean +/- std across 3
stratified folds; a small std means a stable estimate on this split)
```

13. Scientific conclusion

The four learners reach high AUC on the **1,211,875**-row held-out split drawn from **4,847,499** near-balanced **network flows** (no telemetry view is used). The audit below reports the strongest single feature and the duplicate/overlap rates. The ablation decides whether either actually accounts for the score. Per-family recall shows which of the many ToN_IoT attack types are genuinely separated versus carried by easy majority classes.

Validity ledger — read the headline against these printed numbers: Majority-class baseline **accuracy**: **0.5619**. The accuracy column must clear that bar to mean anything. For ROC-AUC the trivial baseline is 0.5, not that figure. Winning learner: **LightGBM** (3-fold CV ROC-AUC **0.9962**). Strongest *single* feature **among the 68 that survive preprocessing**: `Bwd_IAT_Min` at AUC **0.8132**. The ablation refutes a single-feature story. Dropping that feature barely moves the AUC: **0.996751 → 0.996740**. So the separability is **multi-feature**. That reflects how this corpus was generated, not one leaky column. De-duplication lowers the AUC only slightly, to **0.995080**. Repeated rows account for a negligible part of the headline. That verdict is *conservative*, because the de-duplication removes more rows than the capture's own duplication warrants (see the clip caveat below). Data-trust grade: **B**. It is the worse of two independent sub-checks. Single-feature AUC 0.8132 scores **A**. Train/test exact-row overlap 0.114 scores **B**. The overlap check drives the grade, not the single-feature check. That says the split leaks *on the matrix this pipeline built*, not that features are clean; the single-feature check separately scores A. Operational false-positive rate at threshold 0.5: **0.0183**. Worst per-group recalls, exactly as printed: {`mitm`: 0.602, `ddos`: 0.833, `ransomware`: 0.844, `dos`: 0.9, `backdoor`: 0.995, `scanning`: 0.998}. The weakest group sits at **0.602**, which is where detection is thinnest. **Disclosed limitation:** categorical columns are integer-encoded before the split. The encoder therefore sees the test set's category values. On an all-numeric corpus that step is a no-op. The mapping never consults the label, so no *label* information leaks. It is still transductive. A deployed system would need an unseen-category bucket. **How the audit numbers are computed:** overlap is measured on the first 50,000 held-out rows, so read it as a sampled estimate. Each ablation re-splits and refits, so tiny differences are re-split noise. The de-duplication variant keeps the first label when a feature vector appears twice. **All of it is computed on the preprocessed matrix `X`, never on the raw parquet, and that matters here.** The loader ends with `X.clip(-1e15, 1e15)`, which maps every out-of-range value onto one number. CIC-ToN-IoT's `Idle Mean` / `Idle Std` / `Idle Max` / `Idle Min` fields sit partly or wholly above that bound. That is a property of the source file, not something a cell prints. So `Idle Max` is flattened to a constant and dropped, and rows differing only in those fields reach the audit already identical. The exact-duplicate rate **0.307**, the **31%** of rows the de-duplication drops, and the **0.114** train/test overlap are therefore **upper bounds** that include duplication the pipeline manufactured. Grade **B** grades *this pipeline's* split, not CIC-ToN-IoT itself. No decomposition into manufactured versus native duplication is quoted, because no cell here measures the pre-clip rate. **Known defect, disclosed rather than papered over:** a flat ±1e15 clip is the wrong guard for a corpus with fields on that scale. Per-column scaling or a log transform would preserve them. Re-running §10 with the clip removed is the exercise. The code above is left exactly as it ran, so every printed number remains the number this pipeline actually produced. **Scope:** the split is random, not temporal or entity-grouped. Every number above therefore measures in-distribution separability only.

References

1. Moustafa, N. (2021). A new distributed architecture for evaluating AI-based security systems at the edge: Network TON_IoT datasets. *Sustainable Cities and Society*, 72, 102994.
2. Booij, T.M., Chiscop, I., Meeuwissen, E., Moustafa, N. & den Hartog, F.T.H. (2022). ToN_IoT: The Role of Heterogeneity and the Need for Standardization of Features and Attack Types in IoT Network Intrusion Data Sets. *IEEE Internet of Things Journal*, 9(1), 485–496.
3. Sommer, R. & Paxson, V. (2010). Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. *IEEE S&P*.