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

Goal of this notebook

Train and audit detectors on real CTU-13 botnet captures, reporting recall for each malware family.

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.
5. Derive a family label from capture metadata instead of a mislabelled string.

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 3: Vulnerability Assessment** — Learning objective 1 (section 3.1) frames assessment as **evidence grading**, not output collection. The A-F data-trust grade in section 10 is exactly that move, applied to a model score.
- **Chapter 10: Active Deception & Threat Hunting** — Learning objective 6 (section 10.1) places a claim on the **attribution ladder** and corrects for **dependence among rule hits**. The same rule stops us equating one feature with the label.
- **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.

Botnet Detection on the CTU-13 Dataset

Model comparison + validity audit on CTU-13 real botnet captures (≥1M flows)

Abstract: CTU-13 (Garcia et al., 2014) is 13 captures of **real botnet traffic** (Neris, Rbot, Virut, Menti, Sogou, Murlo, NSIS) mixed with real background and normal traffic. We

combine scenarios to ≥1M Argus flows, compare four learners, and audit whether botnet detection is real or a flow shortcut.

1. Research problem

Task: Classify NetFlow-style records as **botnet** vs non-botnet. Unlike KDD99 and many synthetic sets, CTU-13's malicious flows come from *real malware executed in a lab*, with real background traffic. That is a more faithful (and imbalanced) botnet detection problem.

2. Literature review

- **Garcia, Grill, Stiborek & Zunino (2014)** — the CTU-13 dataset and an empirical comparison of botnet-detection methods (Computers & Security).
- **Garcia (2014)** — Stratosphere IPS behavioral models.
- **Sommer & Paxson (2010)** — closed-world ML critique.
- **Beigi et al. (2014)** — flow-feature selection for botnet detection.

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

Reported approach	Known caveat
Garcia et al. (2014) — behavioral methods	evaluated per-scenario; cross-scenario is much harder
Flow-based ML botnet detectors	botnet is rare; background dominates

3. Dataset provenance & honesty caveats

Property	Value
Source	Stratosphere IPS `CTU-13-Dataset` (open, no credentials)
Rows	**1,572,056** Argus flows: every botnet flow within the per-scenario read window plus up to 100,000 non-botnet flows per scenario
Label	`Label` string → **botnet vs non-botnet** (positive = the string contains `Botnet`)
Access	Direct download (1.9 GB tarball)

Honestly: IP/port/time columns are dropped as identifiers; botnet flows are a small minority, so accuracy is base-rate-inflated — per-family recall is the honest metric.

What the negative class actually is: the loader flags a row positive when its `Label` string contains `Botnet`. Everything else becomes class 0. In CTU-13 that remainder is two different things. One is `Normal` traffic, from hosts the authors verified. The other is

`Background` traffic. The dataset authors define Background as "all the rest of traffic that we don't know what it is for sure" (Stratosphere Laboratory, *Datasets Overview*; ref. 2). Background is *unverified*, not *confirmed clean*. It may hide botnet flows nobody identified. So class 0 here means "not labelled botnet". It does not mean "known benign".

Why that matters for the false-positive rate: a flow scored as a false positive may be a real botnet flow sitting in Background. Read the printed operational FPR as an upper bound on analyst nuisance. It is not a measured error rate against verified-clean traffic.

And for training: some class-0 rows are plausibly undetected botnet flows. The model therefore learns from *noisy negatives*. That pushes it toward the majority behaviour and away from whatever those hidden flows look like. Untested hypothesis, stated as a mechanism, not measured here.

Label words in the figures and printed lines: this notebook series shares two plotting variables, `NEG_WORD` and `POS_WORD`. They default to `benign` and `attack`. The CTU-13 loader never overrides them. So the class-balance title, the PCA legend, the confusion-matrix tick labels and the printed line `... benign flagged of ...` all say "benign" and "attack". Read every one of them as **non-botnet** and **botnet**. The cells are not re-run, so the wording stays as printed. Nothing about the mapping changes a number.

Before you run this: getting the data

This notebook downloads its own data on the first run, then caches it. **No Kaggle account and no credentials are needed** - the source is a direct download from Stratosphere IPS, cached under `/tmp/ctu13`.

Check your free disk space first. The tarball is only **1.9 GB**, but it extracts to about **76 GB**, because the archive ships the raw packet captures. This notebook reads none of them: it uses only the 13 `.binetflow` files, **2.5 GB** in total. Make sure you have ~80 GB free before you start, or the extract will fill your disk.

Once the extract finishes you can reclaim almost all of it. The loader only re-downloads when no `.binetflow` file is present, so deleting the captures is safe:

```
find /tmp/ctu13 \( -name '*.pcap' -o -name '*.tar.bz2' \) -delete
```

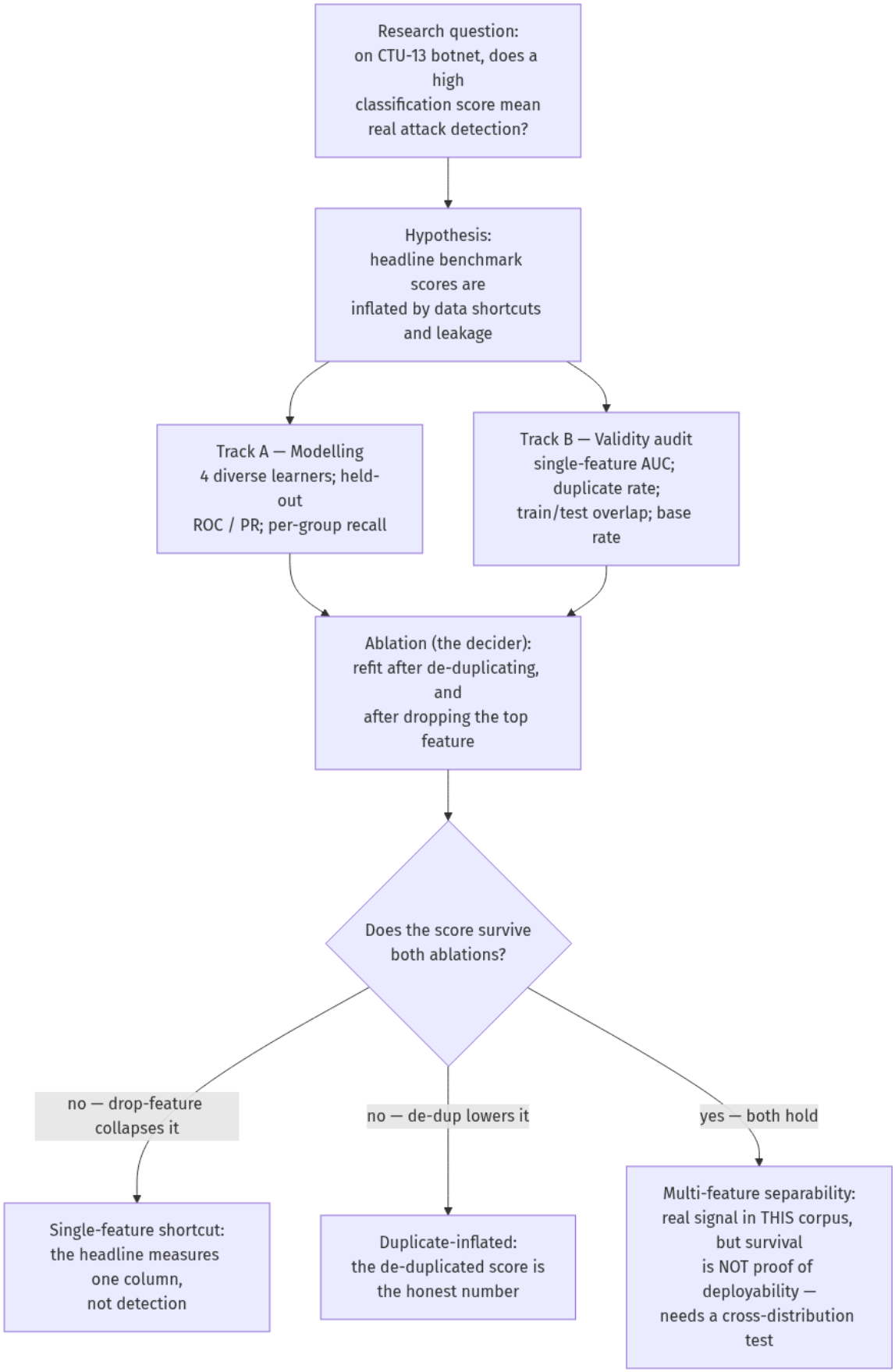
The cache sits under `/tmp`, which macOS clears on reboot. To keep it, move the folder somewhere durable and symlink it back. Do **not** edit the path in the code cell below: that changes a code cell and invalidates the stored outputs you are reviewing.

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.

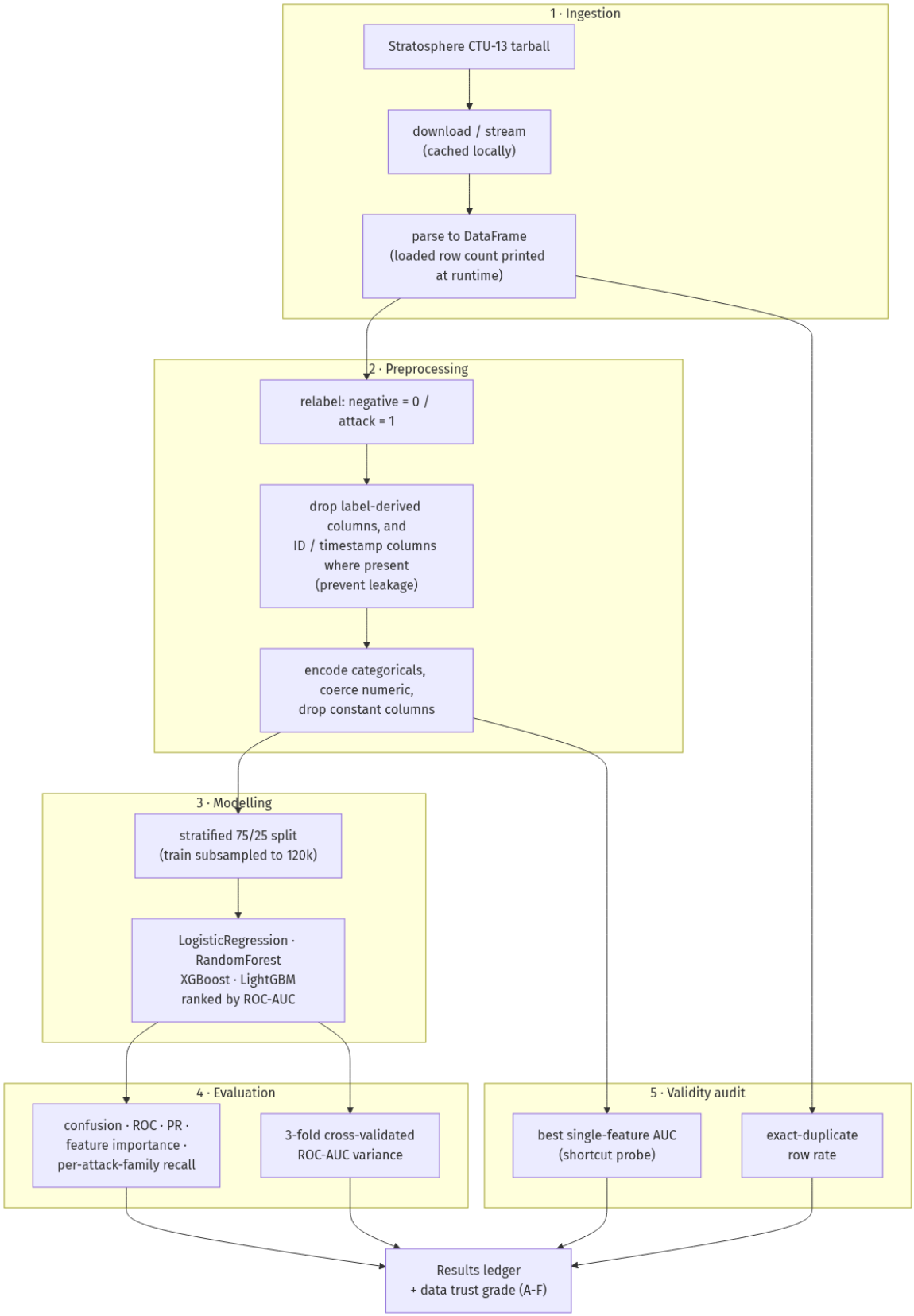
Figure 4.1 — Solution design (methodology).



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, tarfile, urllib.request
# CTU-13 (Garcia et al., 2014): 13 real botnet captures as Argus .binetflow
files.
# Self-contained: download the Stratosphere tarball (1.9 GB, no
credentials) and extract it
# on first run; cached under /tmp/ctu13 thereafter.
CTU_DIR = '/tmp/ctu13'; os.makedirs(CTU_DIR, exist_ok=True)
if not glob.glob(CTU_DIR + '/**/*.binetflow', recursive=True):
    tb = os.path.join(CTU_DIR, 'CTU-13-Dataset.tar.bz2')
    if not os.path.exists(tb):
        print('downloading CTU-13 (~1.9 GB, one-time)...')
        urllib.request.urlretrieve(
            'https://mcfp.felk.cvut.cz/publicDatasets/CTU-13-Dataset/CTU-
13-Dataset.tar.bz2', tb)
    print('extracting...'); tarfile.open(tb).extractall(CTU_DIR)
files = sorted(glob.glob(CTU_DIR + '/**/*.binetflow', recursive=True))
assert files, 'CTU-13 .binetflow files not found after download/extract'
READ_CAP = 1_500_000                                        # rows
scanned per scenario
# Botnet flows are a time-clustered minority within each capture, so we
keep every botnet flow
# *within the first READ_CAP rows of each scenario* and SUBSAMPLE
background. NOTE: because the
# cap truncates long captures, scenarios whose botnet activity starts late
contribute only the
# botnet flows that fall inside the window - so this is a bounded sample,
not the full population;
# it also means the botnet rate below is a bounded-sample rate, not the
natural ~1-2% base rate.
# CTU-13's malware FAMILY is a property of the capture scenario, not of the
flow label string (the
# label only encodes behaviour, e.g. 'flow=From-Botnet-V51-1-UDP-DNS'). The
scenario->malware map is
```

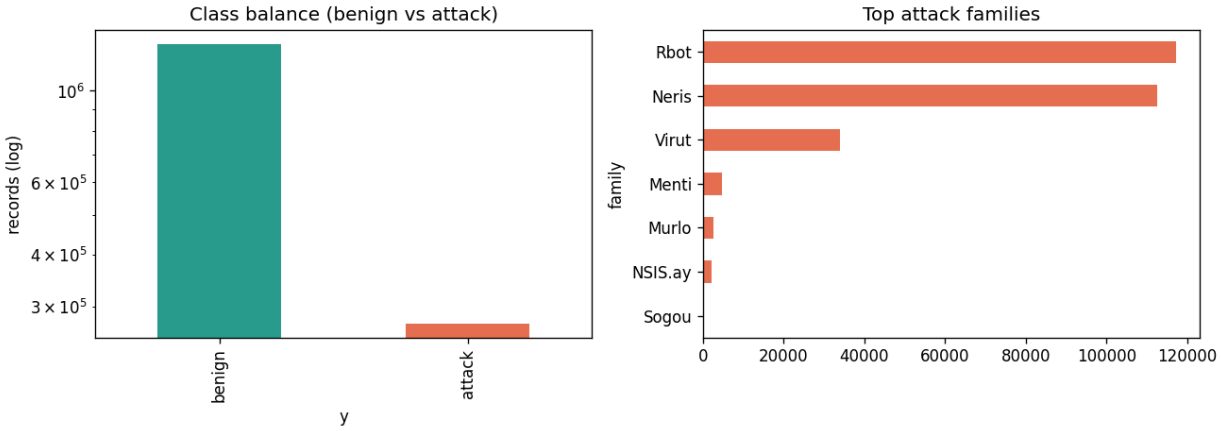
```
# published in Garcia et al. (2014) Table 1; the scenario number is the
directory name.
SCEN_MALWARE = {1:'Neris', 2:'Neris', 3:'Rbot', 4:'Rbot', 5:'Virut',
6:'Menti', 7:'Sogou',
                8:'Murlo', 9:'Neris', 10:'Rbot', 11:'Rbot', 12:'NSIS.ay',
13:'Virut'}
import re as _re
frames = []
for f in files:
    d = pd.read_csv(f, low_memory=False, nrows=READ_CAP)
    d.columns = [str(c).strip() for c in d.columns]
    m = _re.search(r'/(\d{1,2})/[^/]+$', f)                                  #
scenario dir, e.g. .../9/x.binetflow
    scen = int(m.group(1)) if m else 0
    d['scenario'] = scen
    isbot = d['Label'].astype(str).str.contains('Botnet', case=False)  #
botnet vs not
    non = d[~isbot].sample(min((~isbot).sum(), 100_000), random_state=0)  #
bound background
    frames.append(pd.concat([d[isbot], non], ignore_index=True))          #
all botnet + sample
df = pd.concat(frames, ignore_index=True).reset_index(drop=True)
df['y'] = df['Label'].astype(str).str.contains('Botnet',
case=False).astype(int)  # 1 = botnet
# family = the malware actually executed in that capture (botnet rows
only); background stays 'normal'.
df['family'] = np.where(df['y'] == 1,
                        df['scenario'].map(SCEN_MALWARE).fillna('Botnet-
other'),
                        'normal')
print('botnet families recovered:', sorted(set(df.loc[df.y == 1,
'family'])))
assert len(df) >= 1_000_000, f'floor not met: {len(df):,}'
# 'scenario' is capture identity (it maps 1:1 to the malware family) so it
MUST NOT be a feature.
DROP =
['Label','y','family','scenario','StartTime','SrcAddr','DstAddr','Sport','D
port']  # ids/time/labels
feat = [c for c in df.columns if c not in DROP]
from sklearn.preprocessing import LabelEncoder
X = df[feat].copy()
for c in X.select_dtypes(include='object').columns:
    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.loc[:, X.nunique() > 1]; feat = list(X.columns)
y = df['y'].to_numpy()
print(f'loaded {len(df):,} flows x {len(feat)} features; botnet rate
(bounded sample) {y.mean():.4f}')
```

```
botnet families recovered: ['Menti', 'Murlo', 'NSIS.ay', 'Neris', 'Rbot',
'Sogou', 'Virut']
loaded 1,572,056 flows x 9 features; botnet rate (bounded sample) 0.1736
```

7. Exploratory data analysis

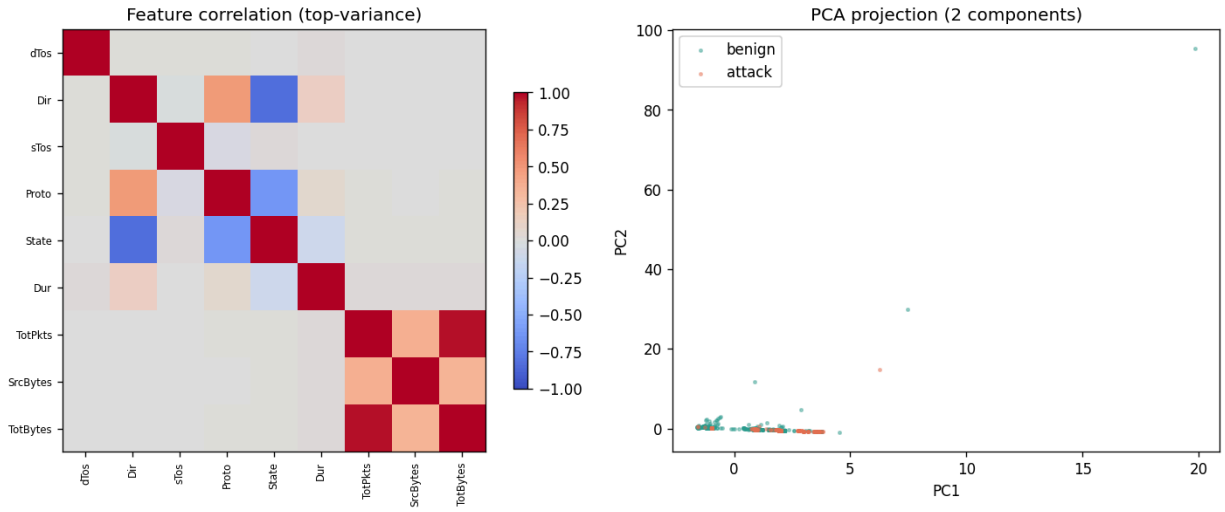
Both figures label the two classes `benign` and `attack`. Read them as **non-botnet** and **botnet**. Section 3 explains why `benign` overstates what the negative class is.

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 1,572,056 rows | trained on 120,000 (stratified subsample) |
held-out 393,014
MAJORITY-CLASS BASELINE accuracy = 0.8264  (any model must beat THIS, not
0.5, to be interesting)
best model: XGBoost
```

Out[5]:

	model	accuracy	roc_auc	train_s
0	XGBoost	0.964182	0.989405	0.3
1	RandomForest	0.968197	0.989373	0.8
2	LightGBM	0.960528	0.987379	1.2
3	LogisticRegression	0.878885	0.804225	0.1
4	MajorityBaseline	0.826400	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.

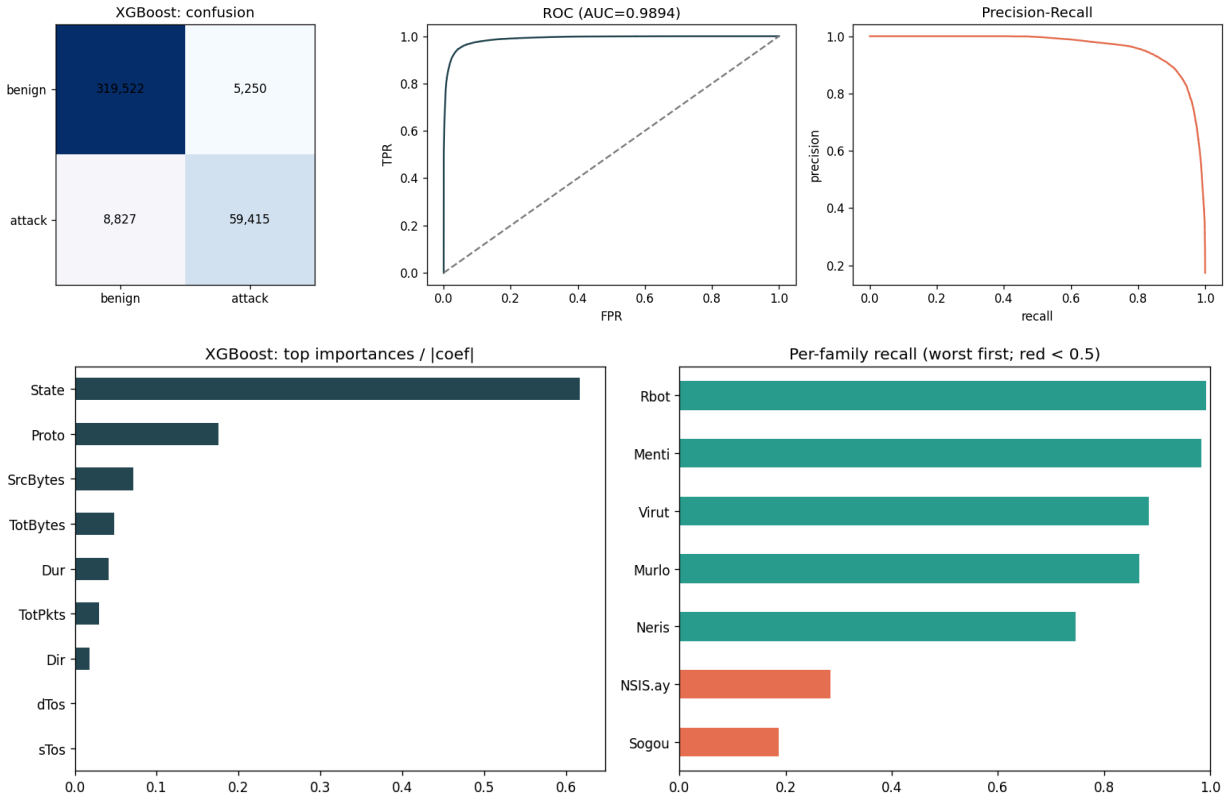
Here `family` is the malware executed in that capture scenario, taken from the published scenario table. Non-botnet rows are grouped as `normal`.

Two wording caveats on the output below. The confusion-matrix axes say `benign` and `attack`; they mean non-botnet and botnet. The printed line says `benign flagged`; those flows are `Background` or `Normal`, and Background was never verified clean.

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 = 0.0162  (5,250 benign flagged of
324,772)
worst per-family recalls: {'Sogou': 0.188, 'NSIS.ay': 0.284, 'Neris': 0.747,
'Murlo': 0.866, 'Virut': 0.885, 'Menti': 0.984}
```

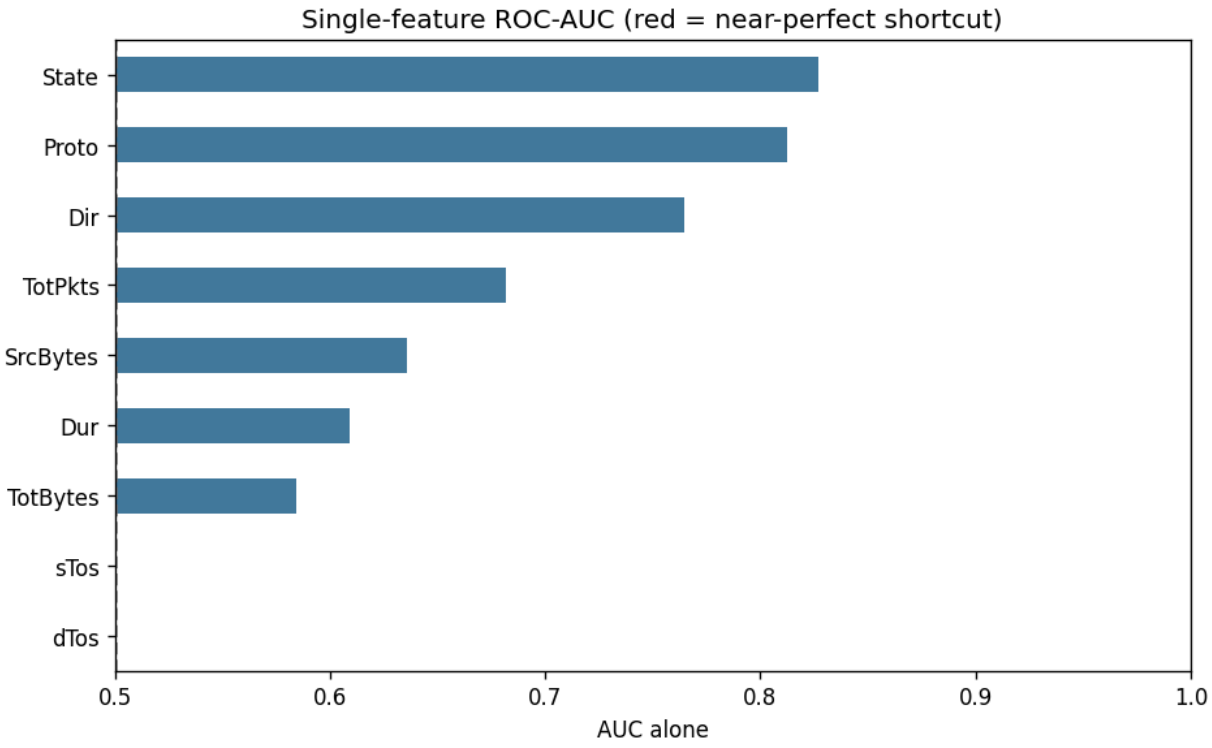
10. Validity audit — is the score real?

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.

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)        #
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 = 0.8271  (feature: State)
   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) = 0.419
TRAIN/TEST exact-row contamination       = 0.353  (single-feat grade A,
contam grade D)
==> data trust grade: D   (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 botnet and non-botnet distributions barely overlap. That is its own generation artifact. The numbers below decide which story is true here, not the prose.

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)	0.989405
1	de-duplicated (42% rows removed)	0.972796
2	shortcut feature dropped (State)	0.988168

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
XGBoost 3-fold CV ROC-AUC = 0.9860 +/- 0.0010  (mean +/- std across 3
stratified folds; a small std means a stable estimate on this split)
```

13. Scientific conclusion

On real botnet captures, aggregate accuracy is flattered by the dominant background/normal traffic; the honest question is per-botnet-family recall and whether flow features generalize across scenarios.

Validity ledger — read the headline against these printed numbers: Majority-class baseline **accuracy**: **0.8264**. The accuracy column must clear that bar to mean anything. For ROC-AUC the trivial baseline is 0.5, not that figure. Winning learner: **XGBoost** (3-fold CV ROC-AUC **0.9860**). Strongest *single* feature: `State` at AUC **0.8271**. The ablation refutes a single-feature story. Dropping that feature barely moves the AUC: **0.989405 → 0.988168**. So the separability is **multi-feature**. That reflects how this corpus was generated, not one leaky column. De-duplication *does* matter here. It lowers the AUC to **0.972796**. So the headline is inflated by repeated rows, and **0.972796** is the honest number. Data-trust grade: **D**. It is the worse of two independent sub-checks. Single-feature AUC 0.8271 scores **A**. Train/test exact-row overlap 0.353 scores **D**. The overlap check drives the grade, not the single-feature check. That says the split leaks, not that features are clean; the single-feature check separately scores A. On this A-best / F-worst scale, a D or F means the headline is optimistic. Treat it as a benchmark number, not a deployment estimate. Operational false-positive rate at threshold 0.5: **0.0162**. Worst per-group recalls, exactly as printed: { `Sogou` : 0.188, `NSIS.ay` : 0.284, `Neris` : 0.747, `Murlo` : 0.866, `Virut` : 0.885, `Menti` : 0.984}. The weakest group sits at **0.188**, so the model misses most of it. That gap, not the aggregate score, is the operationally important result. **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. **Scope:** the split is random, not temporal or entity-grouped. Every number above therefore measures in-distribution separability only. One further scope limit is specific to CTU-13. The negative class mixes verified `Normal` flows with `Background` flows. The dataset authors never verified Background. So the printed false-positive rate of **0.0162** counts alerts on unverified traffic, not confirmed mistakes. The figures and the printed FPR line use the words `benign` and `attack`; they mean non-botnet and botnet.

References

1. García, S., Grill, M., Stiborek, J. & Zunino, A. (2014). An empirical comparison of botnet detection methods. *Computers & Security*, 45, 100–123.
2. García, S. et al. *Stratosphere IPS / Stratosphere Laboratory* (software and dataset project, CTU University, Prague) — https://www.stratosphereips.org. Project resource, not a peer-reviewed publication.
3. Sommer, R. & Paxson, V. (2010). Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. *IEEE S&P*.
4. Biglar Beigi, E., Hadian Jazi, H., Stakhanova, N. & Ghorbani, A. A. (2014). Towards effective feature selection in machine learning-based botnet detection approaches. *2014 IEEE Conference on Communications and Network Security (CNS)*, 247–255. doi:10.1109/CNS.2014.6997492. Commonly short-cited as "Beigi et al."; dblp indexes the first author as Biglar Beigi Samani.