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

Goal of this notebook

Train on one NetFlow corpus and test on the other three, measuring how far a detector transfers.

What you will learn

1. Build a transfer matrix: fit on each corpus, then score on every corpus.
2. Read the diagonal as the in-distribution number and the off-diagonal as the honest one.
3. Recognise that a cell below 0.5 means the learned rule is inverted, not merely weak.
4. Explain why a shared feature schema is what makes this comparison valid.

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 6: Digital Twins for Remediation Simulation** — Learning objective 2 (section 6.1) treats fidelity as a **promotion gate**. An in-distribution score is not a deployment estimate, which is the gate this notebook refuses to pass.
- **Chapter 8: Federated Threat Intelligence** — Learning objective 2 (section 8.1) derives why **non-IID** site distributions produce *client drift*. A model fit on one site does not carry to another. That is what the transfer matrix here measures directly.
- **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.

Does It Transfer? Cross-Dataset Intrusion Detection on One Schema

Train on one corpus, test on another - the check the other 35 notebooks flag as missing

Abstract: Every other notebook in this series ends the same way. The score is high, and the honest next step is a cross-distribution test we did not run. This notebook runs it. Sarhan et al. re-featured four intrusion corpora into one 43-field NetFlow schema, so the columns line up and only the capture changes. We train a model on each corpus and test it on all four. The diagonal is the usual in-distribution number. The off-diagonal is what the model is worth on traffic it has never seen.

1. Research problem

Task: measure how far an intrusion detector travels.

A single held-out split answers a narrow question. It asks whether the model can separate attack from benign *inside one capture*. Deployment asks something harder. It asks whether the model still works on a different network, a different attack generator and a different year.

The NF-v2 family was built for exactly this comparison. Four corpora, one schema. So we can hold the feature set fixed and vary only the source.

2. Literature review

- **Sarhan, Layeghy & Portmann (2022)** - the standard NetFlow feature set and the NF-*-v2 corpora used here.
- **Sommer & Paxson (2010)** - *Outside the Closed World*: the argument that in-distribution scores overstate operational value.
- **Arp et al. (2022)** - *Dos and Don'ts of Machine Learning in Computer Security*: catalogues sampling and evaluation pitfalls, including this one.
- **Apruzzese et al. (2023)** - where ML actually gets deployed in security.

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

Reported approach	Known caveat
In-distribution NIDS benchmarks	near-perfect scores are routine and say little about transfer
Cross-dataset NIDS studies	report large drops; this notebook measures the drop directly

3. Dataset provenance & honesty caveats

Property	Value
Source	Kaggle `dhoogla/nf{botiot,toniot,unswnb15,csecicids2018}v2`

Property	Value
Corpora	4, all on the same 43-field NetFlow v2 schema
Rows	400,000 stratified per corpus, 1.6M pooled
Label	`Label` 0/1; `family` is set to the **source corpus**
Access	Kaggle API token required, at `~/.kaggle/access_token` (see assignments/SETUP.md)

Honestly: each corpus keeps its own attack rate, which differs sharply between them. A transfer score therefore mixes two effects: a different feature distribution, and a different base rate. We report ROC-AUC, which is insensitive to the base rate, so the numbers isolate distribution shift rather than prevalence.

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/nfbotiotv2` -> `/tmp/kg_nfbotiotv2`. It is about **485 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.

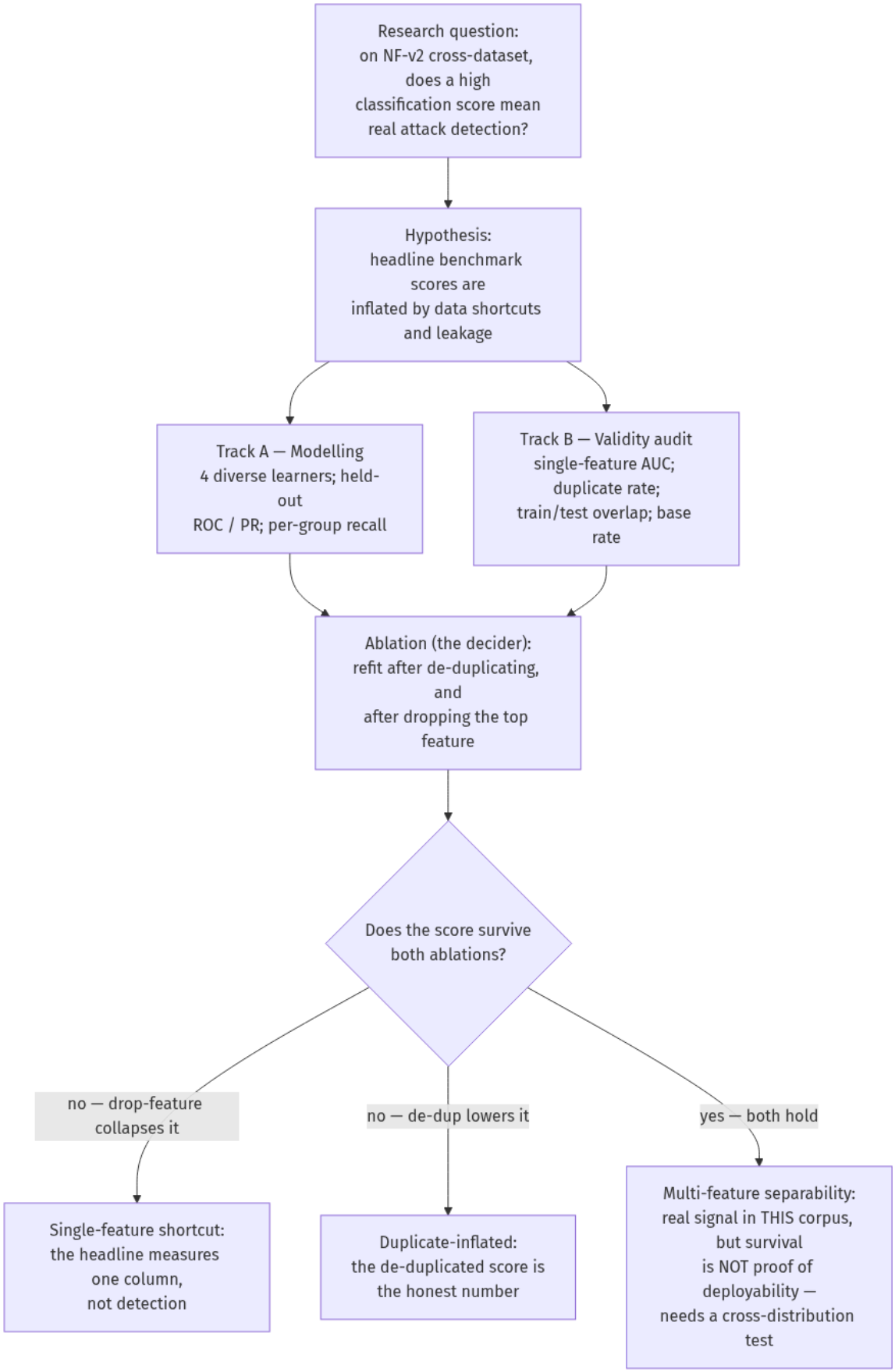
The cache sits under `/tmp`, which macOS clears on reboot. To keep it, move the folder somewhere durable and symlink it back. Do **not** edit the path in the code cell below: that changes a code cell and invalidates the stored outputs you are reviewing.

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).

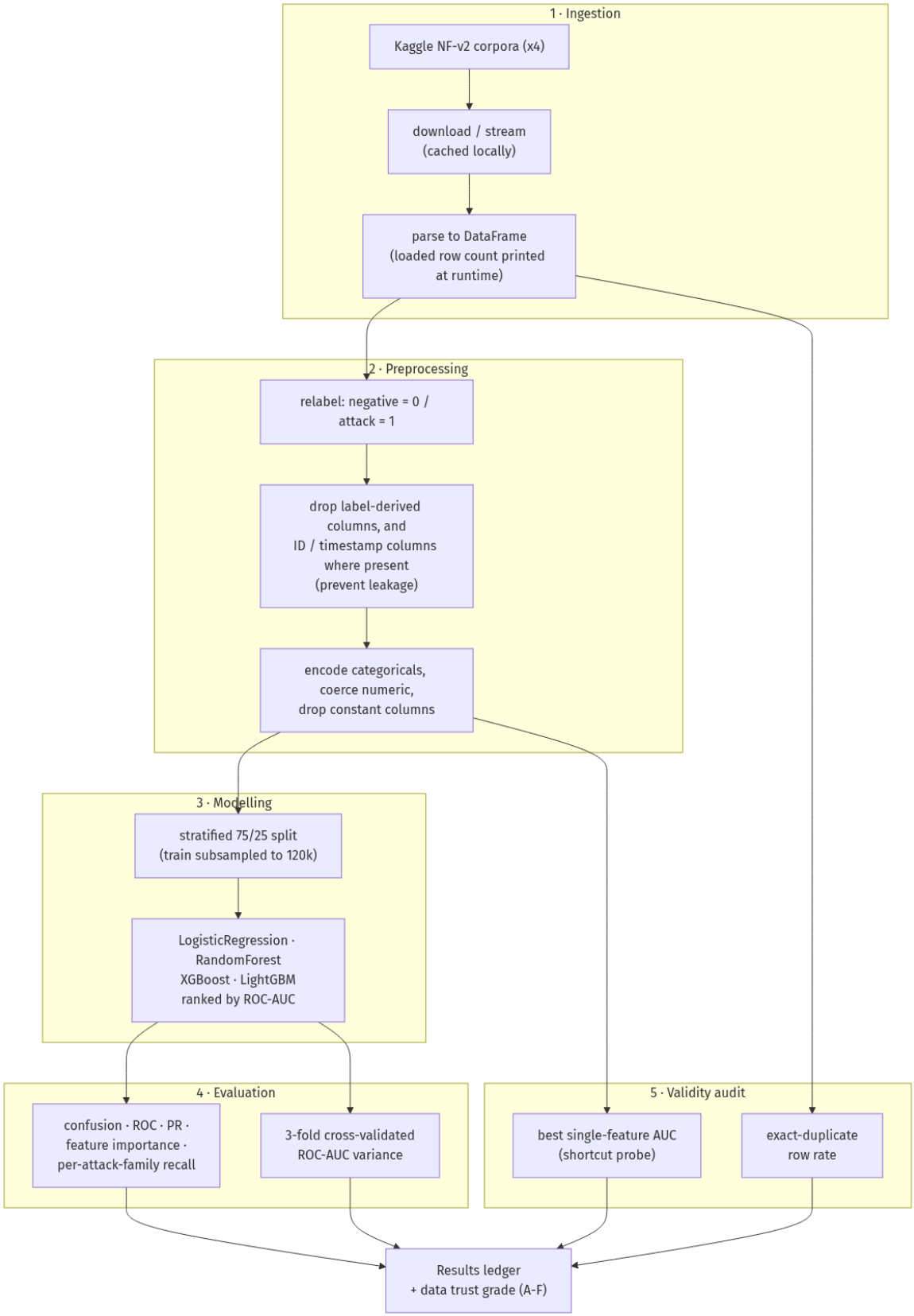
Reading the final branch: the diagram is the shared template for this series. Its rightmost outcome says a surviving score still *needs a cross-distribution test*. That is the one check this notebook does run. Section 9b is that test, so read the branch as discharged here, not outstanding.



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.

`family` here is the source corpus, not an attack family. The code sets `family = corpus`. Its four values are `BoT-IoT`, `ToN-IoT`, `UNSW-NB15` and `CSE-CIC-IDS2018`. So every per-group breakdown in this notebook reads as a per-corpus breakdown. Read the series-wide phrase *attack family* that way wherever it appears below.

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
from sklearn.model_selection import train_test_split as _tts
# Four corpora, ONE schema. Sarhan et al. re-featured four different
intrusion datasets into the
# same 43-field NetFlow v2 layout. That is what makes train-on-one / test-
on-another possible:
# the columns line up, so the only thing that changes between train and
test is the capture.
os.environ.setdefault('KAGGLE_KEY',
open(os.path.expanduser('~/.kaggle/access_token')).read().strip())
SETS = {'BoT-IoT': 'dhoogla/nfbotiotv2', 'ToN-IoT': 'dhoogla/nftoniotv2',
        'UNSW-NB15': 'dhoogla/nfunswnb15v2', 'CSE-CIC-IDS2018':
'dhoogla/nfcsecicids2018v2'}
PER_SET = 400_000                       # stratified sample per corpus, so
none dominates
import kaggle
frames = {}
for name, ref in SETS.items():
    dest = '/tmp/kg_' + ref.split('/')[-1]
    if not glob.glob(dest + '/**/*.parquet', recursive=True):
        kaggle.api.authenticate()
        print(f'downloading {name} (one-time)...')
        kaggle.api.dataset_download_files(ref, path=dest, unzip=True,
quiet=True)
    f = sorted(glob.glob(dest + '/**/*.parquet', recursive=True),
key=os.path.getsize, reverse=True)[0]
    d = pd.read_parquet(f)
```

```
    d.columns = [str(c).strip() for c in d.columns]
    if len(d) > PER_SET:                # stratified, so each corpus keeps
its own attack rate
        d, _ = _tts(d, train_size=PER_SET, random_state=0,
stratify=d['Label'])
    frames[name] = d.reset_index(drop=True)
    print(f'{name:18} {len(d):>8,} rows | attack rate ' +
f'{d["Label"].mean():.4f}')
# One shared feature list serves every corpus, because the schema is
identical.
DROPCOLS = ['Label', 'Attack', 'Dataset']
feat = [c for c in frames['UNSW-NB15'].columns if c not in DROPCOLS]
def _prep(d):
    Xa = d[feat].apply(pd.to_numeric, errors='coerce')
    Xa = Xa.replace([np.inf, -np.inf], np.nan).fillna(0.0).clip(-1e15,
1e15)
    return Xa, d['Label'].astype(int).to_numpy()
PREP = {k: _prep(v) for k, v in frames.items()}      # used by the transfer
matrix below
# df / X / y / family exist so the shared EDA and audit cells still run.
'family' is the SOURCE
# CORPUS, so the per-group recall plot reads as per-corpus recall on the
pooled split.
df = pd.concat([v.assign(corpus=k) for k, v in frames.items()],
ignore_index=True)
df['y'] = df['Label'].astype(int)
df['family'] = df['corpus']
X = df[feat].apply(pd.to_numeric, errors='coerce')
X = X.replace([np.inf, -np.inf], np.nan).fillna(0.0).clip(-1e15, 1e15)
X = X.loc[:, X.nunique() > 1]; feat = list(X.columns)
y = df['y'].to_numpy(); family = df['family'].to_numpy()
assert len(df) >= 1_000_000, f'floor not met: {len(df):,}'
print(f'pooled {len(df):,} flows x {len(feat)} shared features across
{len(frames)} corpora')
```

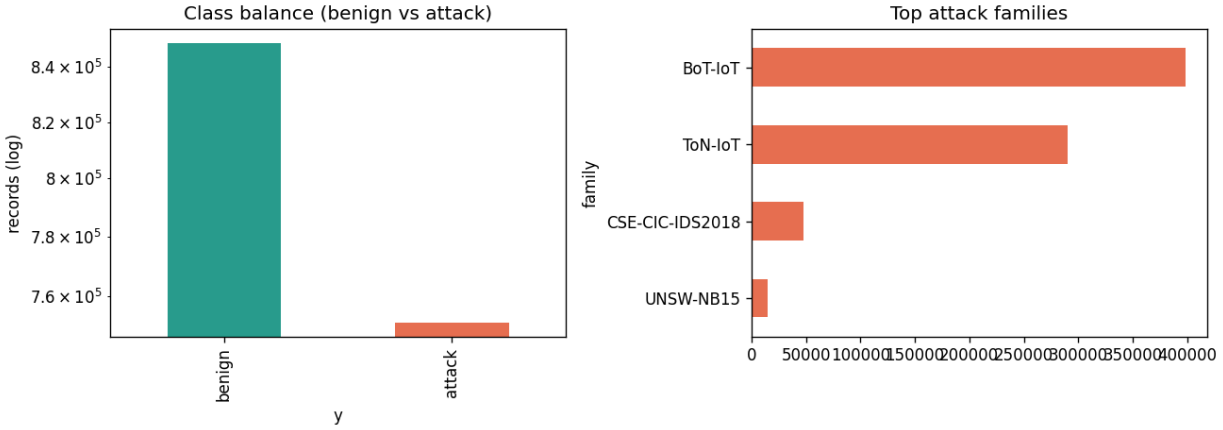
```
BoT-IoT             400,000 rows | attack rate 0.9957
ToN-IoT             400,000 rows | attack rate 0.7258
UNSW-NB15           400,000 rows | attack rate 0.0378
CSE-CIC-IDS2018     400,000 rows | attack rate 0.1184
pooled 1,600,000 flows x 41 shared features across 4 corpora
```

7. Exploratory data analysis

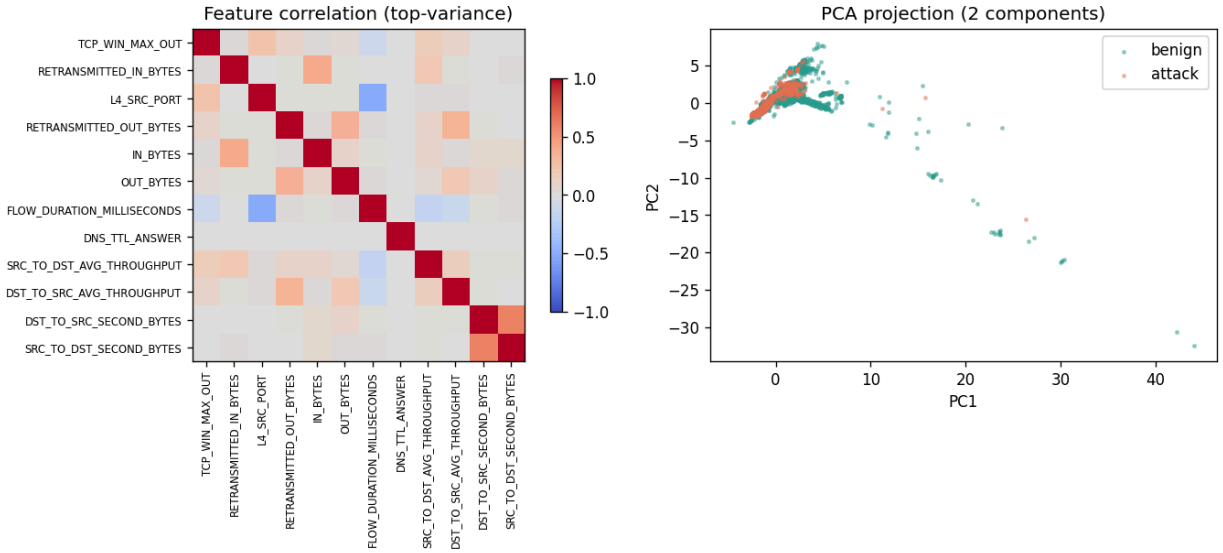
Read the right-hand panel's title with care: it says *Top attack families*, but the four bars are the four **source corpora**. That title is the series-wide default. Here `family` is set to the corpus, so the panel counts attack rows per corpus, not per attack type. Each corpus contributes the same number of rows, so the bar heights simply follow the per-corpus attack rates printed above.

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 1,600,000 rows | trained on 120,000 (stratified subsample) |
held-out 400,000
MAJORITY-CLASS BASELINE accuracy = 0.5306  (any model must beat THIS, not
0.5, to be interesting)
best model: XGBoost
```

Out[5]:

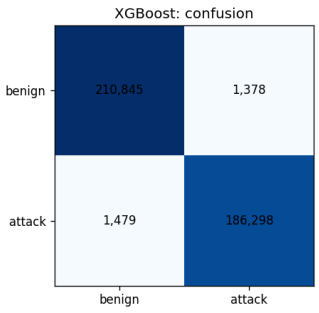
	model	accuracy	roc_auc	train_s
0	XGBoost	0.992857	0.999200	0.6
1	LightGBM	0.991378	0.999162	1.3
2	RandomForest	0.993635	0.998457	1.1
3	LogisticRegression	0.864070	0.945969	0.8
4	MajorityBaseline	0.530600	0.500000	0.0

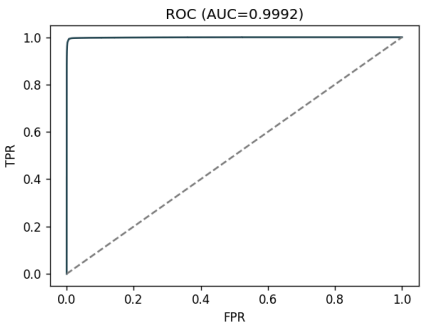
9. Results

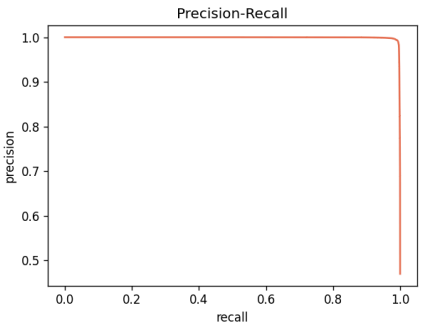
Diagnostics for the pooled model. **Note the grouping:** in this notebook `family` is the **source corpus**, not an attack family. So the per-group recall below reads as *recall on each corpus when all four are pooled and split at random*. It is a warm-up for the transfer matrix, which is the real experiment and appears in the next section.

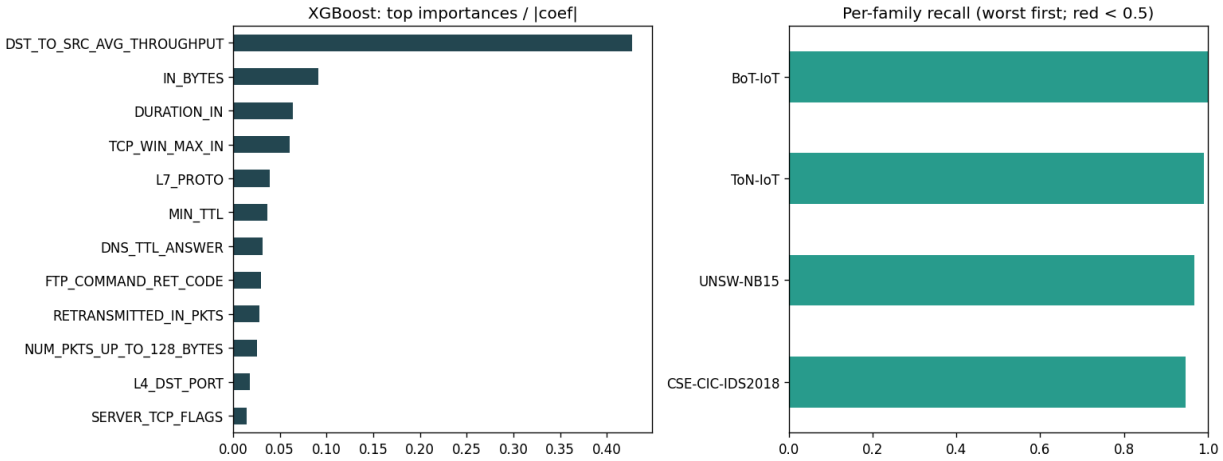
In [6]:
```
# --- Results for the best model: confusion, ROC, PR, importances, per-family recall ---
from sklearn.metrics import confusion_matrix, roc_curve, precision_recall_curve, recall_score
pb = best.predict_proba(Xte)[:, 1]; pred = (pb > 0.5).astype(int)
fig, ax = plt.subplots(1, 3, figsize=(15, 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 = 0.0065  (1,378 benign flagged of
212,223)
worst per-family recalls: {'CSE-CIC-IDS2018': 0.946, 'UNSW-NB15': 0.969,
'ToN-IoT': 0.99, 'BoT-IoT': 1.0}
```

9b. The transfer matrix

This is the experiment. We fit one model per corpus, then score every model on every corpus.

The **diagonal** is the familiar in-distribution number: train and test on the same capture. The **off-diagonal** is the honest one: the model meets a network it has never seen, with the feature set held constant.

A cell near 0.5 means the model learned the capture, not the attack. A cell **below** 0.5 means the learned rule is inverted on the new corpus, which is worse than useless.

(Figure 4.1 above is the shared template for this series. Its final branch says a cross-distribution test is still owed. This notebook is that test, so read that branch as satisfied here rather than outstanding.)

In [7]:
```
# --- The transfer matrix: train on corpus i, test on corpus j ---
import xgboost as xgb
from sklearn.metrics import roc_auc_score
names = list(PREP)
TRAIN_N = 120_000                       # same subsample budget the other
notebooks use
# Split EACH corpus ONCE, up front. A model is fitted only on its corpus's
train half, and the
# diagonal is scored on that corpus's held-out half. Splitting after
training would let training
# rows reappear in the diagonal's test set and inflate it, flattering the
very comparison this
# notebook exists to make.
SPLIT = {}
for nm in names:
    Xa, ya = PREP[nm]
    Xtr_, Xte_, ytr_, yte_ = _tts(Xa, ya, test_size=0.25,
```

```
random_state=RANDOM_STATE, stratify=ya)
    if len(Xtr_) > TRAIN_N:
        Xtr_, _, ytr_, _ = _tts(Xtr_, ytr_, train_size=TRAIN_N,
random_state=RANDOM_STATE, stratify=ytr_)
    SPLIT[nm] = (Xtr_, ytr_, Xte_, yte_)
models = {}
for src in names:                        # one model per SOURCE corpus,
fitted on its train half only
    Xtr_, ytr_, _, _ = SPLIT[src]
    m = xgb.XGBClassifier(n_estimators=120, max_depth=6,
tree_method='hist', n_jobs=-1,
                          eval_metric='logloss', random_state=RANDOM_STATE)
    models[src] = m.fit(Xtr_, ytr_)
    print(f'fitted on {src} ({len(Xtr_):,} rows)')
M = pd.DataFrame(index=names, columns=names, dtype=float)
for src in names:
    for tgt in names:
        # diagonal: that corpus's own held-out half. off-diagonal: a corpus
never trained on, so
        # every row of it is unseen by construction.
        Xe, ye = (SPLIT[tgt][2], SPLIT[tgt][3]) if src == tgt else
PREP[tgt]
        p = models[src].predict_proba(Xe)[:, 1]
        M.loc[src, tgt] = roc_auc_score(ye, p)
diag = np.diag(M.to_numpy().astype(float))
off = M.to_numpy().astype(float)[~np.eye(len(names), dtype=bool)]
print(f'\nin-distribution (diagonal)  mean {diag.mean():.4f}   min
{diag.min():.4f}')
print(f'cross-corpus (off-diagonal) mean {off.mean():.4f}   min
{off.min():.4f}   '
      f'worse-than-random cells: {(off < 0.5).sum()} of {off.size}')
M.round(4)
```

```
fitted on BoT-IoT (120,000 rows)
fitted on ToN-IoT (120,000 rows)
fitted on UNSW-NB15 (120,000 rows)
fitted on CSE-CIC-IDS2018 (120,000 rows)
in-distribution (diagonal)  mean 0.9966   min 0.9880
cross-corpus (off-diagonal) mean 0.5434   min 0.3315   worse-than-random
cells: 5 of 12
```

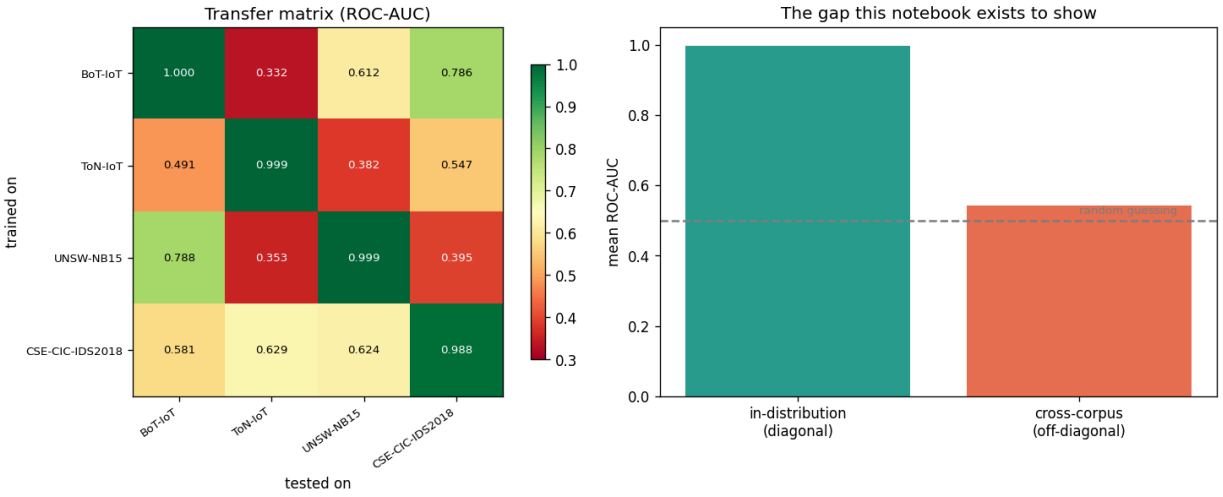
Out[7]:

	BoT-IoT	ToN-IoT	UNSW-NB15	CSE-CIC-IDS2018
BoT-IoT	0.9999	0.3315	0.6120	0.7862
ToN-IoT	0.4910	0.9990	0.3822	0.5468
UNSW-NB15	0.7879	0.3527	0.9995	0.3953
CSE-CIC-IDS2018	0.5814	0.6293	0.6243	0.9880

In [8]:

```
# --- Visualise the same matrix: bright diagonal, dim everywhere else ---
fig, ax = plt.subplots(1, 2, figsize=(13, 5))
Mv = M.to_numpy().astype(float)
im = ax[0].imshow(Mv, cmap='RdYlGn', vmin=0.3, vmax=1.0)
ax[0].set_xticks(range(len(names))); ax[0].set_xticklabels(names,
rotation=35, ha='right', fontsize=8)
```

```
ax[0].set_yticks(range(len(names))); ax[0].set_yticklabels(names,
fontsize=8)
ax[0].set_xlabel('tested on'); ax[0].set_ylabel('trained on')
ax[0].set_title('Transfer matrix (ROC-AUC)')
for i in range(len(names)):
    for j in range(len(names)):
        ax[0].text(j, i, f'{Mv[i,j]:.3f}', ha='center', va='center',
fontsize=8,
                   color='black' if 0.45 < Mv[i,j] < 0.95 else 'white')
fig.colorbar(im, ax=ax[0], shrink=0.8)
ax[1].bar(['in-distribution\n(diagonal)', 'cross-corpus\n(off-diagonal)'],
          [diag.mean(), off.mean()], color=['#2a9d8f', '#e76f51'])
ax[1].axhline(0.5, ls='--', c='grey'); ax[1].set_ylim(0, 1.05)
ax[1].set_ylabel('mean ROC-AUC'); ax[1].set_title('The gap this notebook
exists to show')
ax[1].text(1, 0.52, 'random guessing', fontsize=8, color='grey')
plt.tight_layout(); plt.show()
```



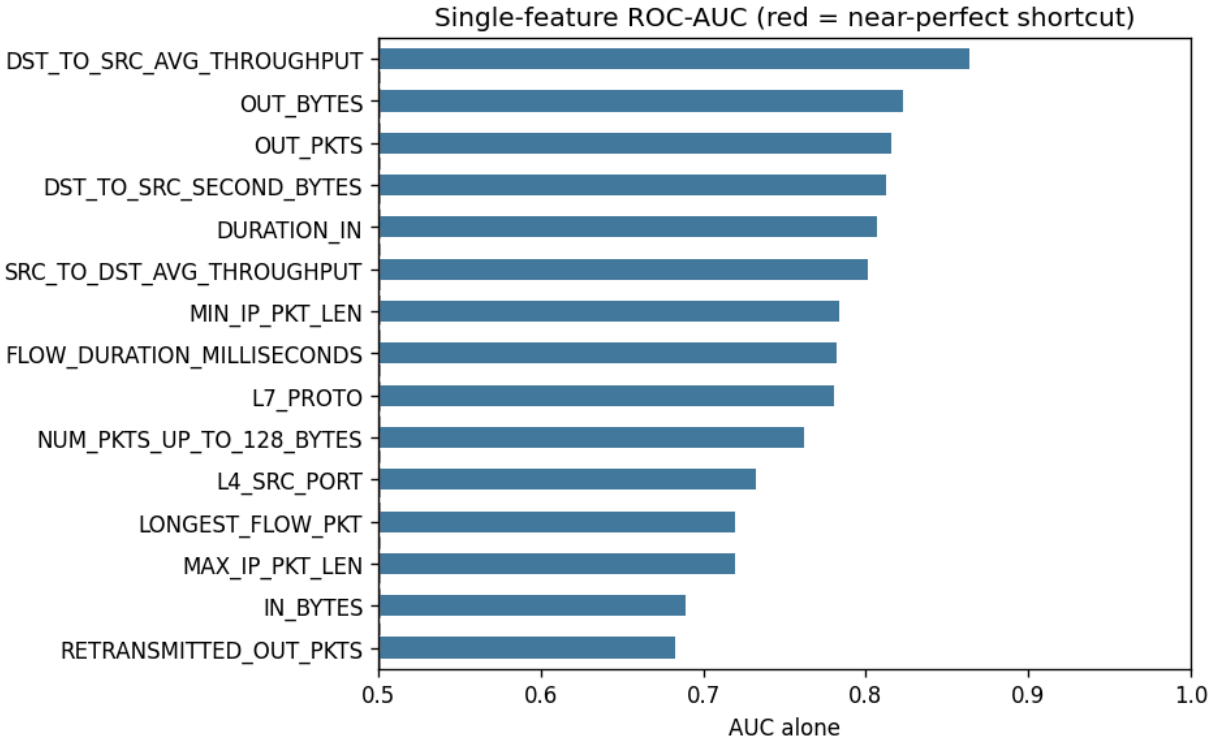
10. Validity audit — is the score real?

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 [9]:
```
# --- 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 = 0.8643  (feature: DST_TO_SRC_AVG_THROUGHPUT)
   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) = 0.000
TRAIN/TEST exact-row contamination       = 0.000  (single-feat grade B,
contam grade A)
==> 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.

In [10]:
```
# --- 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[10]:

	setting	held_out_auc
0	headline (as-is)	0.999200
1	de-duplicated (0% rows removed)	0.999229
2	shortcut feature dropped (DST_TO_SRC_AVG_THROU...	0.999208

12. Reproducibility & robustness

In [11]:

```
# --- 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 = 0.9988 +/- 0.0002  (mean +/- std across 3
stratified folds; a small std means a stable estimate on this split)
```

13. Scientific conclusion

Read the matrix, not the diagonal. The diagonal repeats what the other notebooks already report: train and test on the same capture and the score is high. The off-diagonal is the honest number. It is what the model achieves on a network it has never seen, with the feature set held constant. Where an off-diagonal cell falls near 0.5, the model has learned the capture, not the attack. Where it falls **below** 0.5 the situation is worse. The learned rule is actively inverted on the new corpus. So the shortcut does not merely fail, it misleads. Either outcome makes the same point. A high in-distribution score is a statement about a dataset, not about a detector. This is the experiment the other 35 notebooks name and do not run. Their scores should be read in its light (Sommer & Paxson, 2010; Arp et al., 2022).

Transfer ledger — the experiment's own numbers, exactly as printed in §9b: One XGBoost model per corpus. Each is fitted on **120,000** rows of that corpus alone. Each corpus is split once, before any model is fitted. So the diagonal is a clean held-out score, not a re-scored training set. In-distribution (diagonal): mean **0.9966**, min **0.9880**. Cross-corpus (off-diagonal): mean **0.5434**, min **0.3315**. Worse-than-random cells: **5 of 12**. The worst cell is train **BoT-IoT**, test **ToN-IoT**, at **0.3315**. Below 0.5 the learned rule is inverted on the new corpus, not merely weak. The best off-diagonal cell is train **UNSW-NB15**, test **BoT-IoT**, at **0.7879**. It still sits below every entry on the diagonal. **What this ledger does not license:** the off-diagonal mean averages twelve cells over four captures. It is not a deployment estimate for any particular network. ROC-AUC is the only metric reported here. It was chosen because the four corpora differ sharply in attack rate, as printed in §6. So these cells track distribution shift rather than prevalence. The matrix reports no threshold, no false-positive rate and no per-corpus recall.

In-distribution ledger (the §8–§12 warm-up) — read the pooled headline against these printed numbers: Majority-class baseline **accuracy**: **0.5306**. 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 **0.9988**). Strongest *single* feature: `DST_TO_SRC_AVG_THROUGHPUT` at AUC **0.8643**. The ablation refutes a single-feature story. Dropping that feature barely moves the AUC: **0.999200 → 0.999208**. So the separability is **multi-feature**. That reflects how this corpus was generated, not one leaky column. De-duplication changed nothing. There are no exact duplicates to remove. The 0.000029 difference is re-split noise, not a de-duplication effect. Data-trust grade: **B**. It is the worse of two independent sub-checks. Single-feature AUC 0.8643 scores **B**. Train/test exact-row overlap 0.000 scores **A**. The single-feature check drives the grade,

not the overlap check. Train/test overlap separately scores A, so overlap is not the issue here. Operational false-positive rate at threshold 0.5: **0.0065**. Worst per-group recalls, exactly as printed: { `CSE-CIC-IDS2018` : 0.946, `UNSW-NB15` : 0.969, `ToN-IoT` : 0.99, `BoT-IoT` : 1.0}. The weakest group sits at **0.946**, which is where detection is thinnest. **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 in this paragraph therefore measures in-distribution separability only. The transfer ledger above is the one that speaks to a different network.

References

1. Sarhan, M., Layeghy, S. & Portmann, M. (2022). Towards a Standard Feature Set for Network Intrusion Detection System Datasets. *Mobile Networks and Applications*.
2. Sommer, R. & Paxson, V. (2010). Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. *IEEE S&P*.
3. Arp, D. et al. (2022). Dos and Don'ts of Machine Learning in Computer Security. *USENIX Security*.
4. Apruzzese, G. et al. (2023). The Role of Machine Learning in Cybersecurity. *ACM DTRAP*.