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

Goal of this notebook

Train and audit fraud detectors on a synthetic transaction corpus, comparing ranking quality against decisions at a threshold. The audit is the point: the headline score here does **not** survive it, and section 13 explains exactly which check missed and why.

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. Separate ranking quality (AUC) from decision quality at a threshold.
6. Recognise a **missingness shortcut**: when a loader fills absent values with a sentinel, a tree can split on *"was this field even recorded?"*. Meanwhile, a per-feature AUC audit sees nothing at all. This notebook contains a live example.

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 4: Attack Graph Analytics** — Learning objective 5 (section 4.1) separates a **one-at-a-time** perturbation from the smallest perturbation that reverses a ranking, and warns the first **overstates stability**. Dropping only the top feature and refitting is exactly that weaker test, so read it as a floor.
- **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.

Large-Scale Financial Transaction Fraud Detection

Model comparison + validity audit on Financial-Fraud (5M synthetic) (≥1M records, via Kaggle)

Abstract: A financial-fraud dataset of **5,000,000 transactions**, with rich behavioural features (amount, velocity, geo-anomaly and spending-deviation scores). This notebook analyses the **first 1,500,000** — the loader read cap. Every figure below describes that slice, whose fraud rate is ~2.26% versus ~3.59% across the full file. We compare four learners and audit the scores — with an explicit caveat about the dataset's synthetic provenance. **Result up front:** the audit passes, the headline does not. Section 13 shows, from the notebook's own printed figures, that the winning model's top feature carries no ranking signal at all. It also shows that the ablation removed a feature the model barely used. The reported grade A is therefore a statement about the audit's coverage rather than about the data.

1. Research problem

Task: Flag fraudulent transactions among a large majority of legitimate ones using behavioural and contextual features. Fraud is rare and adversarial; the analytical question is whether the engineered anomaly scores encode the label (a shortcut) or genuine behaviour.

2. Literature review

- **Dal Pozzolo et al. (2015)** — learning with imbalanced fraud data; probability calibration.
- **Bahnsen et al. (2016)** — feature engineering for credit-card fraud detection.
- **Sommer & Paxson (2010)** — closed-world evaluation caveats (apply to fraud ML too).

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

Reported approach	Known caveat
Gradient-boosted trees on engineered fraud features	engineered anomaly scores can leak the label
Imbalanced-learning baselines	accuracy meaningless at <1% fraud

3. Dataset provenance & honesty caveats

Property	Value
Source	Kaggle `aryan208/financial-transactions-dataset-for-fraud-detection`

Property	Value
Rows	5,000,000 in the full file; **1,500,000 analysed here** (leading rows, per the loader read cap). Base rates differ between the two: ~3.59% fraud overall vs ~2.26% in the analysed slice
Label	`is_fraud` (bool); family `transaction_type`
Access	Kaggle API token required

Honestly — critical: this is a **synthetic, community-generated** dataset with **no peer-reviewed origin**. It is a large-scale *methods sandbox*, not a validated benchmark; engineered columns like `geo_anomaly_score` may encode the label. Account IDs, IP, device hash, timestamp and `fraud_type` are dropped to prevent leakage; watch the ablation and single-feature audit especially closely here.

A second, subtler defect — and the one the audit below does not catch. One field, `time_since_last_transaction`, is by construction only defined when a prior transaction exists. It is therefore legitimately absent for some rows. The loader's `X = ... .fillna(0.0)` converts every such absence into the sentinel value **0**. That single line changes the question the model can answer. If absence is associated with the label, then "this field was never recorded" becomes a label proxy. A tree can split that proxy out in one node. Section 10's per-feature ROC-AUC check is structurally incapable of seeing it. That is because ROC-AUC scores a column's *ordering*, and a sentinel need not sit at either end of that ordering. Section 13 works through the printed evidence that this is what happened here, and gives the one-line check that would settle it.

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 `aryan208/financial-transactions-dataset-for-fraud-detection` -> `/tmp/kg_financial-transactions-dataset-for-fraud-detection`. It is about **759 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:

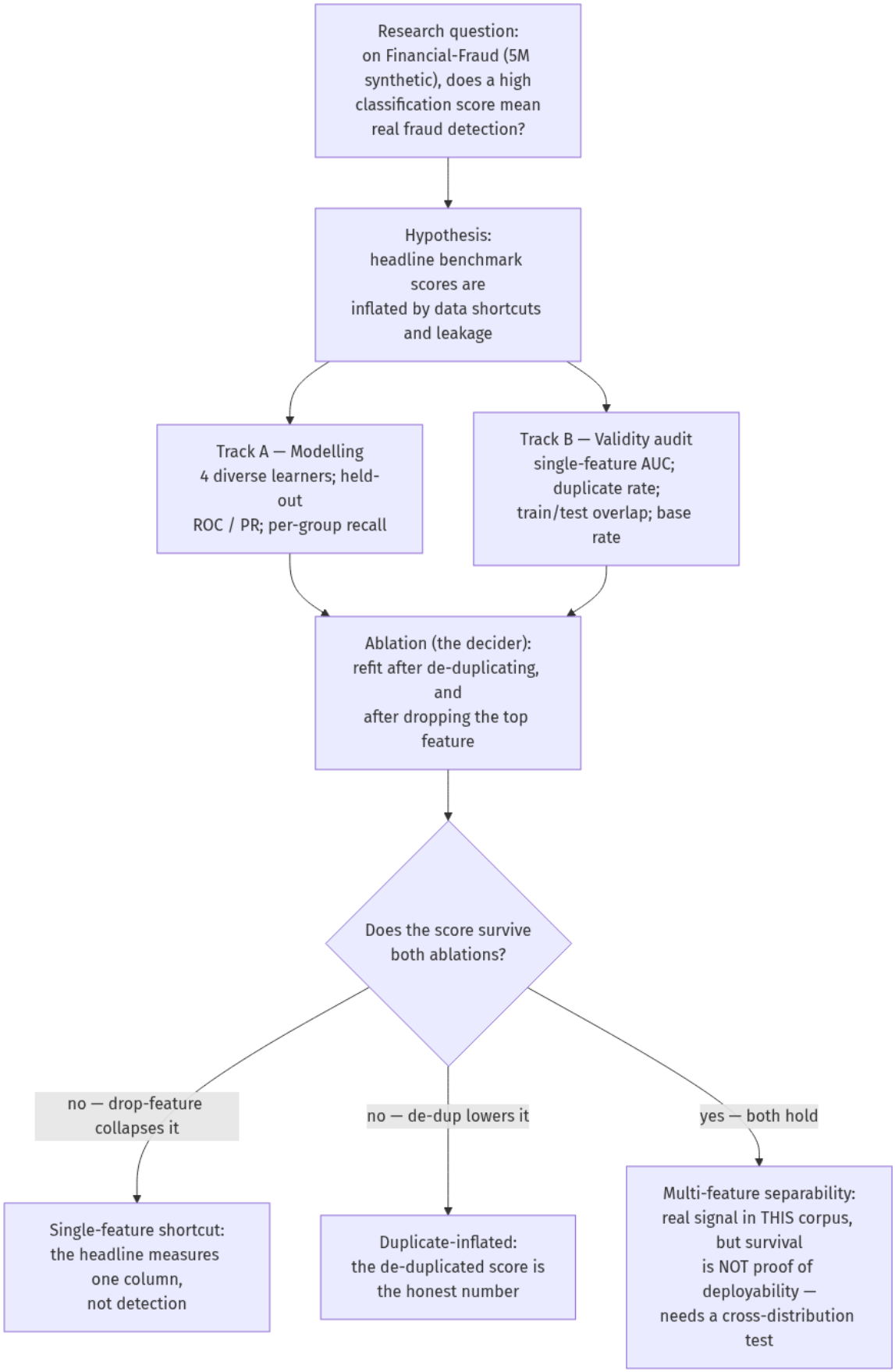
- `FileNotFoundError: ~/.kaggle/access_token` - you created `kaggle.json` but not the key file. Run the command above.
- `401 Unauthorized` - the key is wrong, or a trailing newline crept in.
- `403 Forbidden` - open the dataset page on Kaggle while signed in, accept its terms, then re-run the cell.

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.

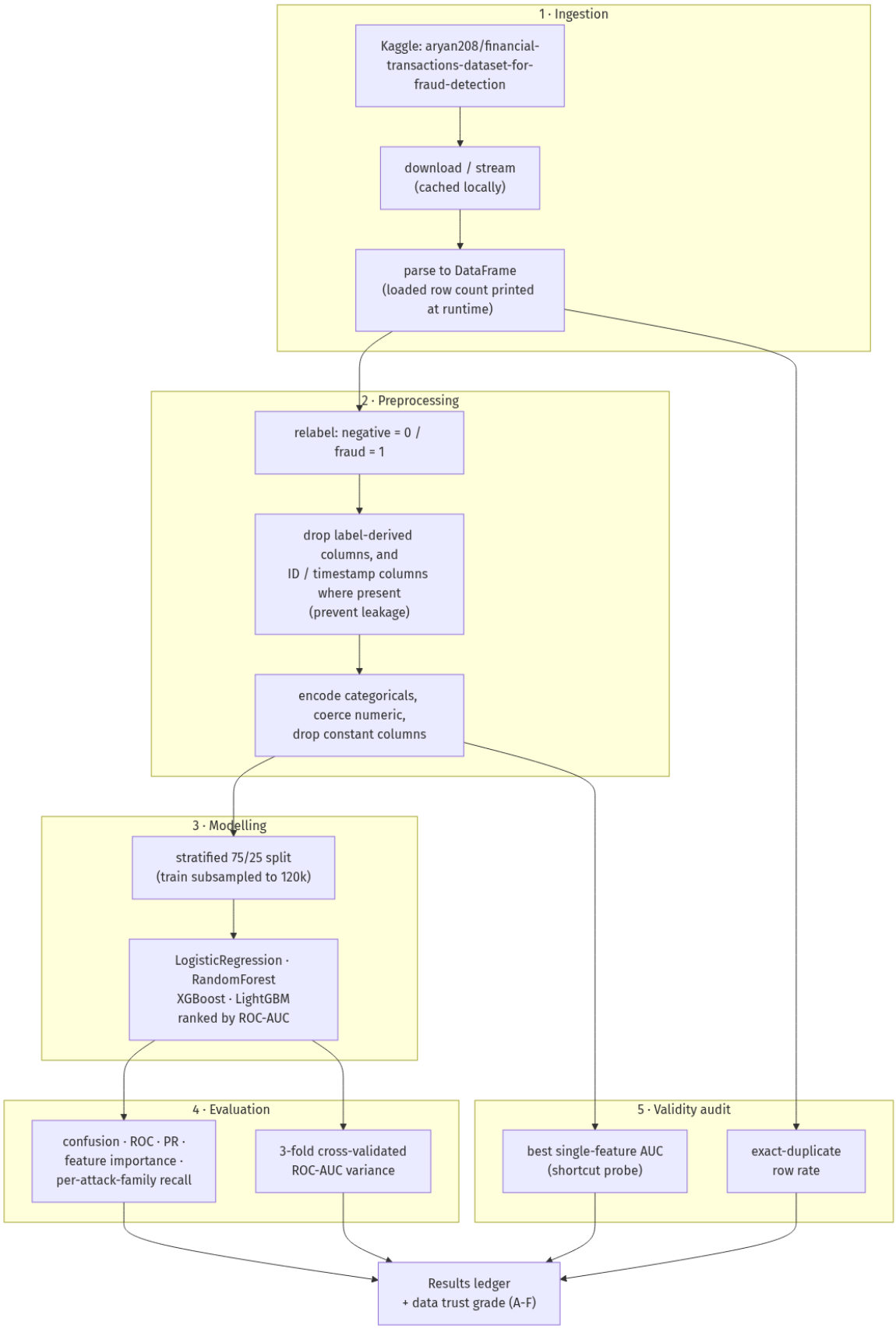
Figure 4.1 — Solution design (methodology).



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 = 'aryan208/financial-transactions-dataset-for-fraud-detection'; 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 + '/**/*.csv', recursive=True))
df = pd.concat([pd.read_csv(f, low_memory=False, nrows=NROWS) 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 = 'is_fraud'; FAMILY = 'transaction_type'
df['y'] = (df[LABEL].astype(str).str.strip().str.lower() !=
'false').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(['transaction_id',
'sender_account', 'receiver_account', 'ip_address', 'device_hash',
'timestamp', 'fraud_type']))  # 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()
```

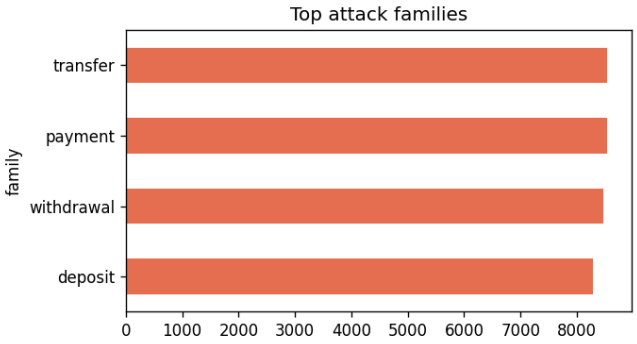
```
idlike = [c for c in X.select_dtypes(include='object').columns if
X[c].nunique() > 0.5*len(X)]
X = X.drop(columns=idlike)                                    # drop
ID/timestamp-like leaky columns
for c in X.select_dtypes(include='object').columns:          # encode
remaining categoricals
    X[c] = 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 = 'legitimate', 'fraud'                   # class names for
plots
print(f'loaded {len(df):,} rows x {len(feat)} features; positive rate
{y.mean():.4f}')
```

```
loaded 1,500,000 rows x 9 features; positive rate 0.0226
```

7. Exploratory data analysis

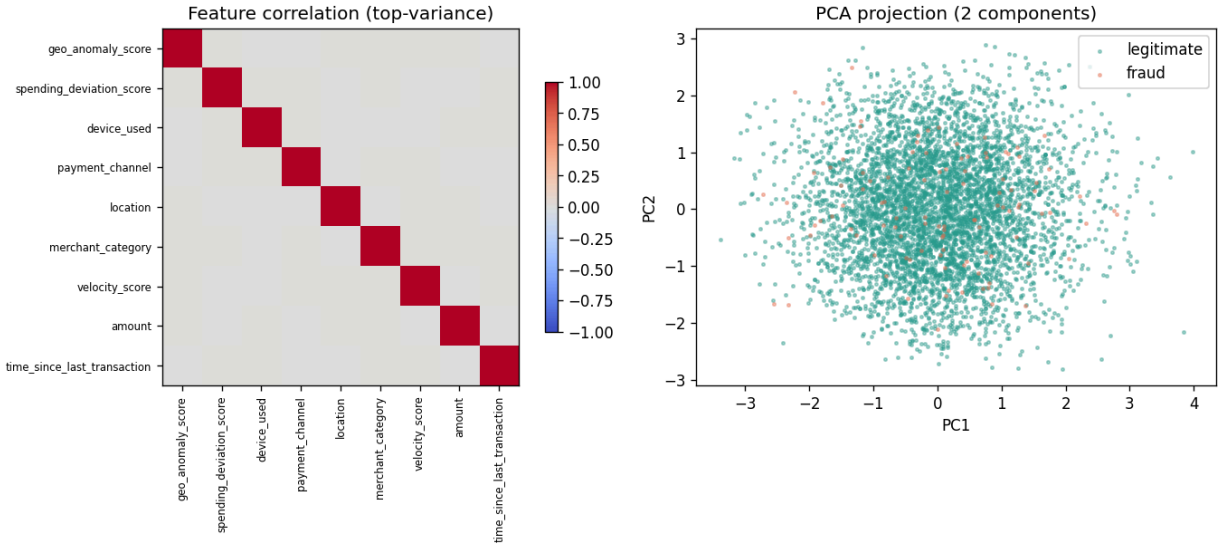
In [3]:

```
# --- 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 1,500,000 rows | trained on 120,000 (stratified subsample) |
held-out 375,000
MAJORITY-CLASS BASELINE accuracy = 0.9774  (any model must beat THIS, not
0.5, to be interesting)
best model: LightGBM
```

Out[5]:

	model	accuracy	roc_auc	train_s
0	LightGBM	0.977445	0.748990	1.1
1	XGBoost	0.977419	0.746834	0.4
2	RandomForest	0.977448	0.725762	0.7
3	LogisticRegression	0.977448	0.501583	0.1
4	MajorityBaseline	0.977400	0.500000	0.0

9. Results

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

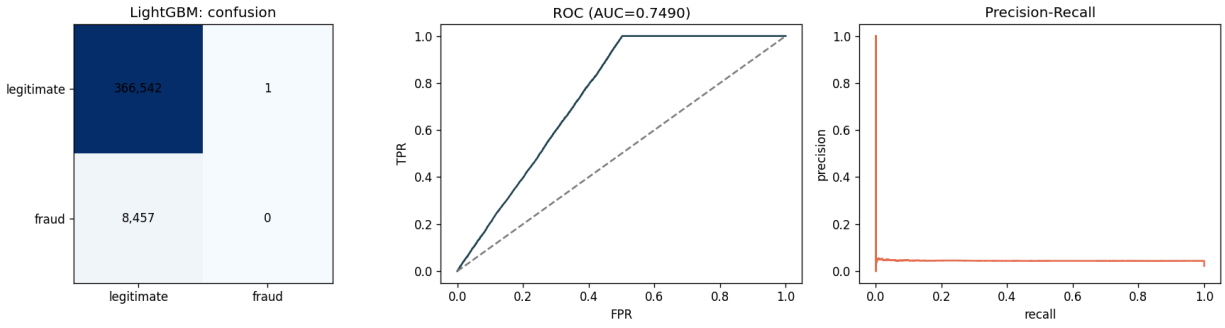
The grouping comes from whatever the loader put in `family`. It is *not* always an attack taxonomy. On the intrusion corpora it is the attack family. On the fraud and malware corpora it is a transaction type, a merchant category or a malware category. On binary corpora it collapses to the positive class.

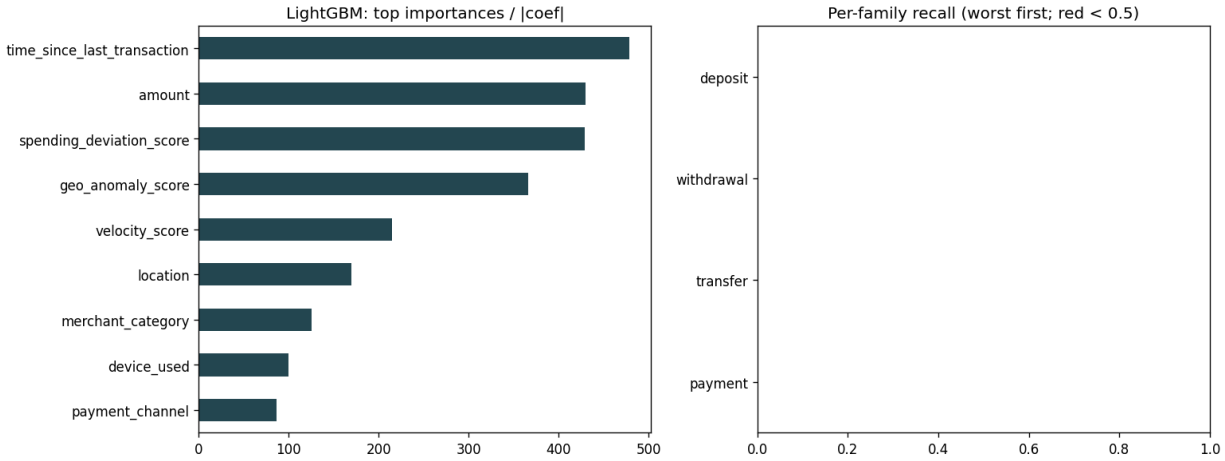
Read it accordingly. Where the groups are genuinely rare classes, they reveal whether detection is real. The dominant flood classes do not.

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 = 0.0000  (1 benign flagged of 366,543)
worst per-family recalls: {'payment': 0.0, 'transfer': 0.0, 'withdrawal':
0.0, 'deposit': 0.0}
```

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.

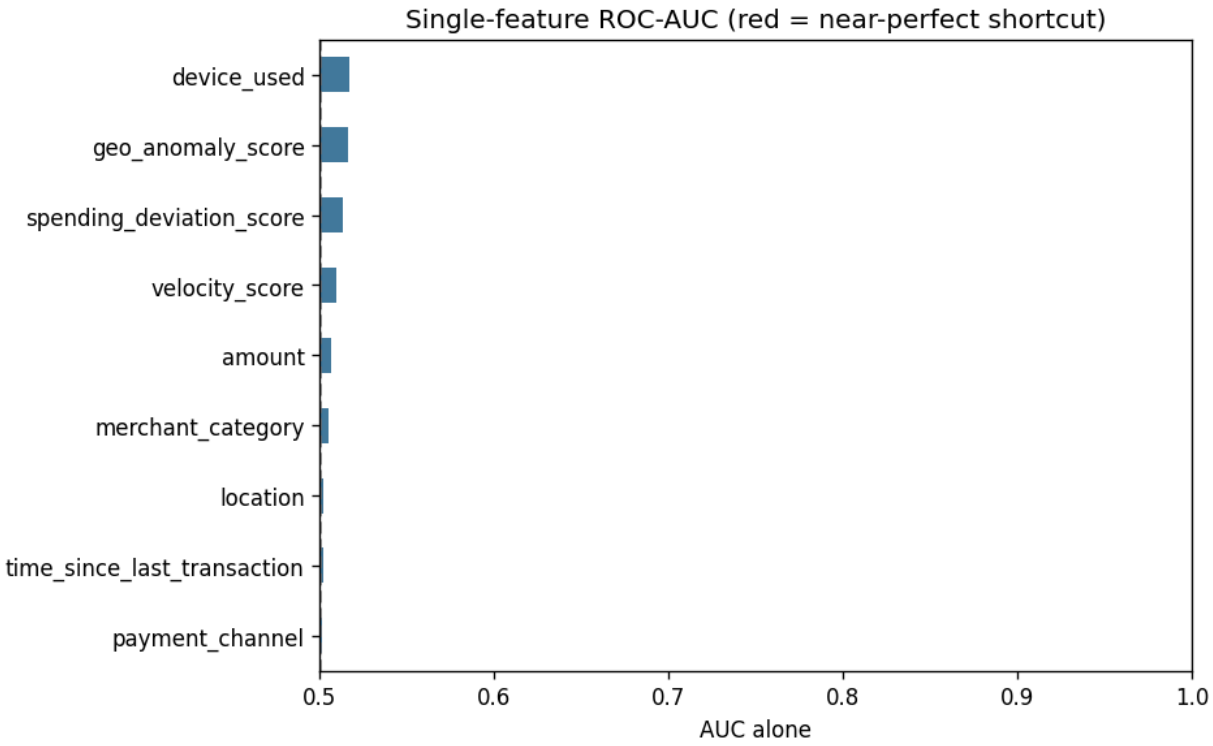
Known blind spot — read the grade narrowly, and on this corpus do not accept it. Diagnostic (a) is a **rank** statistic: it asks whether sorting rows by a column also sorts the label. It is therefore blind to signal that lives in a **split at one particular value** rather than in the ordering. That is not a hypothetical here, because the loader ends with `X.fillna(0.0)`. Every absent value becomes the sentinel 0, and a tree can carve that sentinel out even when the column's overall ordering carries nothing. So an A on this diagnostic means only *"no column ranks the label"* — never *"no shortcut exists"*.

The check the grade cannot do for you: put the chart printed below next to the importance chart printed in section 9. Suppose the winning model's **top** feature sits near the **bottom** of the single-feature AUC chart. Then, whatever the model is using, this diagnostic is not measuring it. In that case the grade is uninformative rather than reassuring. Look at the two charts before reading section 11.

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 = 0.5171  (feature: device_used)
   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 A,
contam grade A)
==> data trust grade: A   (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 — a stable AUC has three different explanations, and you must tell them apart. If the AUC **collapses**, the headline was a contamination/shortcut artifact. If it **barely moves**, then either:

1. the classes really are separable by *many* redundant features — common on *simulated* corpora, and its own generation artifact rather than a vindication; or
2. **the ablation dropped the wrong column.** The feature removed here is whichever one won diagnostic (a), and diagnostic (a) ranks by ROC-AUC. If the real shortcut is a split-based one that a rank statistic cannot see, this cell obediently deletes a feature the model hardly used. Then nothing moves, and the stability means nothing at all.

Case 2 is what happens in this notebook. Before you read a flat ablation as good news, find the dropped feature in the section 9 importance chart. An ablation that perturbs the model's eighth-most-used feature has not tested the headline. 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)	0.748990
1	de-duplicated (0% rows removed)	0.748990
2	shortcut feature dropped (device_used)	0.748274

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 = 0.7459 +/- 0.0038  (mean +/- std across 3
stratified folds; a small std means a stable estimate on this split)
```

13. Scientific conclusion

Verdict: INCONCLUSIVE. Do not read the 0.748990 as fraud detection. The audit in section 10 returns grade **A**, and every arithmetic step producing that grade is correct. The grade is still not evidence that the headline is real, because the two failure modes it enumerates are not the failure mode this corpus contains. That gap — between an audit that passes and a result that holds — is the entire lesson of this notebook.

Validity ledger — the printed numbers

- **Majority-class baseline accuracy 0.9774.** Every model lands on it: LightGBM 0.977445, XGBoost 0.977419, RandomForest and LogisticRegression 0.977448. The accuracy column carries no information whatsoever. For ROC-AUC the trivial baseline is 0.5, not that figure.
- **Winning learner: LightGBM**, held-out ROC-AUC **0.748990**; 3-fold CV ROC-AUC **0.7459 +/- 0.0038**.
- **Strongest single feature:** `device_used` at AUC **0.5171**. Every one of the nine features scores below 0.52 on its own.
- **Exact-duplicate row rate 0.000**; **train/test exact-row overlap 0.000**. There is nothing to de-duplicate, so the de-duplication ablation returns the identical **0.748990** — that is a no-op, not a finding.
- **Ablation, dropping `device_used` : 0.748990 → 0.748274.**
- **Data-trust grade A** (single-feature sub-check A, contamination sub-check A).
- **Operational false-positive rate at threshold 0.5: 0.0000** — **1** benign transaction flagged out of **366,543**.
- **Per-transaction-type recall, exactly as printed:** { `payment` : 0.0, `transfer` : 0.0, `withdrawal` : 0.0, `deposit` : 0.0}.

Why the tempting reading of that ledger is wrong

The tempting reading is: no single feature is near-sufficient, dropping the strongest one changes almost nothing, so the separability must be genuine multi-feature behavioural signal. Three pieces of the notebook's own printed output refute it.

1. **The ablation dropped a feature the model barely used.** It removes whichever feature wins diagnostic (a) — here `device_used`, which the section 9 importance chart ranks **eighth of nine** for the winning model. Moving from 0.748990 to 0.748274 after deleting the eighth-most-used feature is not a test of the headline; it is a measurement of how little that feature mattered.
2. **The model's top feature has no ranking signal at all.** The section 9 importance chart puts `time_since_last_transaction` **first**. The section 10 chart puts that same column near the **bottom**, essentially on the 0.5 chance line. A column cannot be both the most-used feature and rank-uninformative unless the model is splitting it at a *specific value* rather than using its order. That is precisely the case diagnostic (a) is built not to see.
3. **The spread across learners says the same thing.** LogisticRegression, which after scaling can only use each column monotonically, scores **0.501583** — a coin flip. The three split-based learners reach **0.725762** to **0.748990**. Signal that only split-based learners can reach, concentrated in a column whose ordering carries nothing, is the signature of a threshold artifact, not of behavioural fraud signal.

The most likely explanation — stated as a hypothesis this notebook does not test

`time_since_last_transaction` is only defined when a prior transaction exists, so it is absent for some rows, and the loader's `X.fillna(0.0)` turns every absence into the sentinel 0. If absence is associated with the label, then *"this field was never recorded"* is a near-free label proxy. A tree finds it in one split, a rank statistic cannot see it, and the de-duplication and drop-`device_used` ablations both leave it completely untouched.

This notebook never measures that, so it is not reported as a result: It is the check a student should run before trusting any number above:

```
df.groupby('y')['time_since_last_transaction'].apply(lambda s:
s.isna().mean())
```

If those two missingness rates differ sharply, then the 0.748990 is bookkeeping about which rows happened to have a recorded prior transaction rather than fraud detection. Note that the section 10 rubric would **still** return A even then. It grades one feature's AUC against fixed thresholds (0.85, 0.95, 0.99). An artifact therefore need only be worth about as much as the headline — nowhere near 1.0 — to pass straight through it. The question a validity audit should ask is not *whether any one feature is near-perfect*, but *whether any one artifact accounts for the whole headline*. The honest fix here is to drop the column. Better still, add an explicit `was_missing` indicator and re-run the whole audit against it. The shortcut then becomes a feature you can see instead of one hiding inside a fill value.

Corrected grade and the transferable lesson

Corrected data-trust grade: not established. Grade A is an accurate statement about the two checks section 10 runs — no column *ranks* the label, and no held-out row duplicates a training row. Neither check covers sentinel-encoded missingness. "The audit found nothing" and "there is nothing to find" are different claims, and only the first is supported.

The transferable lesson is about audit design, not about this dataset: **a validity audit is only ever as strong as the failure modes it enumerates.** A per-feature rank statistic is structurally blind to anything a tree accomplishes with a single split. An ablation that is *steered by* that rank statistic inherits the same blind spot. It will keep deleting the wrong column and keep reporting that nothing changed. Two checks agreeing is not corroboration when they share an assumption.

And none of this is a working detector in any case

At threshold 0.5 the model flags **1** transaction out of **366,543** benign ones and achieves **0.0** recall on *every* transaction type. Its deployed behaviour is "predict legitimate, always" — which is exactly the 0.9774 baseline. Ranking quality and decision quality are different things; here both fail, and the per-group recall breakdown, not the aggregate AUC, is what makes that visible.

Disclosed limitations

Categorical columns are integer-encoded before the split, so the encoder sees the test set's category values. The mapping never consults the label, so no *label* information leaks. The step is still transductive, and a deployed system would need an unseen-category bucket. On an all-numeric corpus that step is a no-op. Train/test 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 small differences are re-split noise rather than effects — for scale, the winning model's own 3-fold CV spread is +/- 0.0038. The de-duplication variant keeps the first label when a feature vector appears twice. Models are fitted on a stratified subsample of 120,000 rows drawn from the 1,500,000 loaded, and scored on 375,000 held-out rows. The split is random, not temporal or entity-grouped, so every number above measures in-distribution separability only.

References

1. Dal Pozzolo, A. et al. (2015). Calibrating probability with undersampling for unbalanced classification. *IEEE SSCI*.
2. Bahnsen, A.C. et al. (2016). Feature engineering strategies for credit card fraud detection. *Expert Systems with Applications*, 51.
3. Sommer, R. & Paxson, V. (2010). Outside the Closed World. *IEEE S&P*.