05_01_26_v3_clean

January 5, 2026

1 Next-Year Financial Distress Prediction (Compustat Annual Panel) — Reproducible ML Pipeline

Goal. Predict the probability that a firm is in *financial distress* in fiscal year **t+1** using accounting (and permitted market) information available at fiscal year **t**.

Important scope note. The outcome is an **engineered distress proxy** (high leverage / balance-sheet stress), not a realized legal default or bankruptcy. The notebook is therefore a **predictive measurement and decision-support pipeline**, not a causal identification design.

1.1 Notebook structure (Data Science Lifecycle — 10 phases)

1. Problem Definition & Setup
2. Data Collection & Panel Integrity
3. Data Cleaning & Missingness Handling (leakage-aware)
4. Exploratory Data Analysis (EDA)
5. Feature Engineering & Target Construction
6. Preprocessing for Modeling (train-only fitting)
7. Model Selection & Training (7A Logit; 7B Trees)
8. Model Evaluation & Diagnostic Monitoring
9. Decision Support Layer (events, lift, scenarios, cost/decision curves)
10. Results Summary, Guardrails, and Replication Artifacts

    This organization mirrors the course lifecycle guidance and the project's technical review action items (see provided PDF and technical report).

1.2 How to run (replication package convention)

1. Place `data.csv` in the project root (or update `CONFIG["DATA_PATH"]` in Section 1).

2. Keep `Variables.xlsx` (variable dictionary) alongside the notebook for automatic documentation.
3. Run **Kernel → Restart & Run All**.

The notebook creates an `outputs/` folder containing: - a predictions export (`predictions.csv`), - configuration and threshold tables, - model summary tables suitable for an appendix, - figures saved as PNG for paper workflow.

1.3 1. Problem Definition & Setup

1.3.1 1.1 Prediction target, success metrics, and decision objective

- **Target (supervised label):** `target_next_year_distress` = *lead* of an NA-aware distress proxy within firm.

- **Primary performance metrics (out-of-sample):**
  - ROC-AUC (ranking quality),
  - PR-AUC (class imbalance),
  - Brier score (probability accuracy / calibration).
- **Decision objective (screening):** convert predicted PDs into a review policy using:
  - **misclassification costs** (`COST_FN`, `COST_FP`) and
  - **capacity constraints** (screen top `CAPACITY_PCT` percent of firms).

This is a *risk scoring* workflow: calibrated probabilities and operational interpretability matter more than headline accuracy.

1.3.2 1.2 Configuration, determinism, and library versions

[211]:
```
# Core numerics
import os
import sys
import math
import json
import warnings
from pathlib import Path
from dataclasses import dataclass, asdict

import numpy as np
import pandas as pd

# ML / metrics
from sklearn.model_selection import ParameterGrid
from sklearn.preprocessing import StandardScaler
from sklearn.impute import KNNImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    roc_auc_score,
    average_precision_score,
    brier_score_loss,
```

```
    confusion_matrix,
    precision_recall_curve,
    roc_curve,
)
from sklearn.calibration import calibration_curve
from sklearn.isotonic import IsotonicRegression

# Stats / inference
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.stats.sandwich_covariance import cov_cluster, cov_cluster_2groups
from scipy import stats

# Trees / explainability
import xgboost as xgb

import matplotlib.pyplot as plt
from IPython.display import display

warnings.filterwarnings("ignore")

# ----------------------------
# Determinism
# ----------------------------
SEED = 42
np.random.seed(SEED)

# ----------------------------
# Configuration (edit here)
# ----------------------------
CONFIG = {
    # Data inputs
    "DATA_PATH": "data.csv",
    "VARIABLES_XLSX_PATH": "Variables.xlsx",

    # Temporal splitting via label_year = fyear + 1
    "TRAIN_CUTOFF_LABEL_YEAR": 2022,   # label_year <= cutoff -> train/val pool; later -> test
    "VAL_YEARS": 1,                    # number of last label years inside the train pool used as validation

    # Missingness / imputation
    "KNN_K": 25,
    "IMPUTE_LO_Q": 0.01,
    "IMPUTE_HI_Q": 0.99,
```

```
    # Preprocessing
    "WINSOR_LO_Q": 0.01,
    "WINSOR_HI_Q": 0.99,

    # Logit hyperparameter search
    "LOGIT_C_GRID": [0.01, 0.1, 1.0, 10.0],

    # Tree model (XGBoost) parameters (conservative / regularized)
    "XGB_PARAMS": {
        "max_depth": 4,
        "min_child_weight": 5,
        "subsample": 0.8,
        "colsample_bytree": 0.8,
        "eta": 0.05,
        "reg_lambda": 10.0,
        "reg_alpha": 0.0,
        "objective": "binary:logistic",
        "eval_metric": "aucpr",
        "tree_method": "hist",
        "seed": SEED,
    },
    "XGB_NUM_BOOST_ROUND": 5000,
    "XGB_EARLY_STOPPING": 100,

    # Decision policy parameters (costs + capacity)
    "COST_FN": 10.0,
    "COST_FP": 1.0,
    "CAPACITY_PCT": 0.20,  # screen top 20% by PD as a capacity policy

    # Outputs
    "OUTPUT_DIR": "outputs",
    "FIG_DIR": "figures",
}

Path(CONFIG["OUTPUT_DIR"]).mkdir(parents=True, exist_ok=True)
Path(CONFIG["FIG_DIR"]).mkdir(parents=True, exist_ok=True)

print("CONFIG (key parameters):")
for k in␣
  ↪["DATA_PATH","TRAIN_CUTOFF_LABEL_YEAR","VAL_YEARS","KNN_K","WINSOR_LO_Q","WINSOR_HI_Q","COS
  ↪
    print(f"  {k}: {CONFIG[k]}")
print("\nPython:", sys.version.split()[0])
print("pandas:", pd.__version__)
print("numpy:", np.__version__)
```

```
CONFIG (key parameters):
  DATA_PATH: data.csv
```

```
  TRAIN_CUTOFF_LABEL_YEAR: 2022
  VAL_YEARS: 1
  KNN_K: 25
  WINSOR_LO_Q: 0.01
  WINSOR_HI_Q: 0.99
  COST_FN: 10.0
  COST_FP: 1.0
  CAPACITY_PCT: 0.2

Python: 3.13.5
pandas: 2.3.1
numpy: 2.2.5
```

1.3.3 1.3 Helper utilities (robust ratios, transforms, and reporting)

[212]:
```python
def signed_log1p(x: pd.Series) -> pd.Series:
    """Signed log1p transform: sign(x) * log1p(|x|). Preserves zero and sign,␣
↪stabilizes tails."""
    x = pd.to_numeric(x, errors="coerce")
    return np.sign(x) * np.log1p(np.abs(x))

def safe_divide(numer: pd.Series, denom: pd.Series, denom_floor: float = None)␣
↪-> pd.Series:
    """Safe divide with optional denominator floor for stability. Returns float␣
↪with NaN where undefined."""
    numer = pd.to_numeric(numer, errors="coerce")
    denom = pd.to_numeric(denom, errors="coerce")
    if denom_floor is not None:
        denom = denom.where(denom.abs() >= denom_floor, other=np.sign(denom).
↪replace(0, 1) * denom_floor)
    out = numer / denom
    out = out.replace([np.inf, -np.inf], np.nan)
    return out

def ensure_nullable_float(s: pd.Series) -> pd.Series:
    """Convert to pandas nullable Float64 to enable NA-aware comparisons␣
↪(returns <NA> instead of False)."""
    return pd.to_numeric(s, errors="coerce").astype("Float64")

def winsorize_train_bounds(x: pd.Series, lo: float, hi: float) -> tuple[float,␣
↪float]:
    """Return winsorization bounds computed on *training* observed values."""
    x = pd.to_numeric(x, errors="coerce")
    x_obs = x.dropna()
    if len(x_obs) == 0:
        return (np.nan, np.nan)
    return (float(x_obs.quantile(lo)), float(x_obs.quantile(hi)))
```

```
def apply_bounds(x: pd.Series, lo: float, hi: float) -> pd.Series:
    x = pd.to_numeric(x, errors="coerce")
    if np.isnan(lo) or np.isnan(hi):
        return x
    return x.clip(lower=lo, upper=hi)

def compute_smd(train: pd.Series, test: pd.Series) -> float:
    """Standardized mean difference (SMD): (mu_train - mu_test)/pooled_sd."""
    a = pd.to_numeric(train, errors="coerce").dropna()
    b = pd.to_numeric(test, errors="coerce").dropna()
    if len(a) < 2 or len(b) < 2:
        return np.nan
    mu_a, mu_b = a.mean(), b.mean()
    sd_a, sd_b = a.std(ddof=1), b.std(ddof=1)
    pooled = np.sqrt(0.5*(sd_a**2 + sd_b**2))
    return float((mu_a - mu_b) / pooled) if pooled > 0 else np.nan

def logit(p: np.ndarray, eps: float = 1e-6) -> np.ndarray:
    p = np.clip(p, eps, 1-eps)
    return np.log(p/(1-p))

def sigmoid(z: np.ndarray) -> np.ndarray:
    return 1/(1+np.exp(-z))

def print_df(df: pd.DataFrame, n: int = 10, name: str = None):
    if name:
        print(f"\n{name} (top {n} rows):")
    display(df.head(n))
```

1.4 2. Data Collection & Panel Integrity

1.4.1 2.1 Load variable dictionary (for documentation)

We load the provided variable dictionary (`Variables.xlsx`) to: - validate required Compustat mnemonics exist in the data file, - generate appendix-ready variable tables.

This step **does not** transform the modeling data.

[213]:
```
vars_path = Path(CONFIG["VARIABLES_XLSX_PATH"])
if vars_path.exists():
    var_dict = pd.read_excel(vars_path, sheet_name=0)
    var_dict.columns = [c.strip() for c in var_dict.columns]
    print(f"Loaded variable dictionary with {len(var_dict)} rows from:␣
 ↪{vars_path}")
    display(var_dict.head(12))
else:
```

```
    var_dict = pd.DataFrame(columns=["Variable","Two-word␣
↪Description","Category"])
    print(f"WARNING: variable dictionary not found at {vars_path}. Continuing␣
↪without it.")
```

```
Loaded variable dictionary with 89 rows from: Variables.xlsx
```

	Variable	Two-word Description	Category
0	aco	Other Current	Balance Sheet
1	act	Current Assets	Balance Sheet
2	ao	Other Assets	Balance Sheet
3	aoloch	Asset/Liability Δ	Cash Flow
4	ap	Accounts Payable	Balance Sheet
5	apalch	AP & Accrued Δ	Cash Flow
6	aqc	Acquisitions CF	Cash Flow
7	at	Total Assets	Balance Sheet
8	caps	Capital Surplus	Balance Sheet
9	capx	CapEx	Cash Flow
10	ceq	Common Equity	Balance Sheet
11	che	Cash Equivalents	Balance Sheet

1.4.2 2.2 Load raw data (no imputation or transformations)

[214]:
```
data_path = Path(CONFIG["DATA_PATH"])
df_raw = pd.read_csv(data_path, low_memory=False)
print(f"Loaded data from {data_path} with shape {df_raw.shape}")

display(df_raw.head())
```

```
Loaded data from data.csv with shape (75005, 89)
```

	gvkey	datadate	fyear	indfmt	datafmt	consol	ismod	\
0	1004	2015-05-31	2014	INDL	STD	C	1	
1	1019	2014-12-31	2014	INDL	STD	C	1	
2	1045	2014-12-31	2014	INDL	STD	C	1	
3	1050	2014-12-31	2014	INDL	STD	C	1	
4	1072	2015-03-31	2014	INDL	STD	C	1	

	conm	aco	act	…	txach	txbcof	\
0	AAR CORP	101.6	954.1	…	0.0	0.0	
1	AFA PROTECTIVE SYSTEMS INC	541.0	23369.0	…	NaN	0.0	
2	AMERICAN AIRLINES GROUP INC	1260.0	12112.0	…	0.0	0.0	
3	CECO ENVIRONMENTAL CORP	17424.0	142967.0	…	-1164.0	923.0	
4	AVX CORP	108638.0	1744552.0	…	NaN	474.0	

	txdc	txditc	txp	txt	xi	xido	xidoc	xint
0	-79.8	104.60	0.00	-28.5	0.0	64.7	-133.7	26.5
1	-676.0	1635.00	132.00	1058.0	0.0	0.0	0.0	203.0
2	346.0	NaN	0.00	330.0	0.0	0.0	0.0	915.0

```
3   -3183.0  26365.00  405.00  3137.0  0.0   0.0    0.0  3138.0
4 -58387.0      5.77    4.45 -7272.0  0.0   0.0    0.0   978.0

[5 rows x 89 columns]
```

1.4.3 2.3 Enforce panel identifiers, types, sorting, and deduplication

[215]:
```
df = df_raw.copy()

# Stable firm identifier
if "gvkey" not in df.columns:
    raise ValueError("Required identifier column `gvkey` not found in the␣
 ↪dataset.")
df["firm_id"] = df["gvkey"].astype(str)

# Fiscal year
if "fyear" not in df.columns:
    raise ValueError("Required time column `fyear` not found in the dataset.")
df["fyear"] = pd.to_numeric(df["fyear"], errors="coerce").astype("Int64")

# Optional datadate parsing (kept as metadata; not used for splitting)
if "datadate" in df.columns:
    df["datadate"] = pd.to_datetime(df["datadate"], errors="coerce")

# Remove firm-year duplicates (keep-last rule, audit count)
pre_n = len(df)
dup_mask = df.duplicated(subset=["firm_id","fyear"], keep=False)
n_dups = int(dup_mask.sum())
if n_dups > 0:
    print(f"Found {n_dups} duplicated firm-year rows. Applying keep-last rule.")
    df = df.sort_values(["firm_id","fyear","datadate"] if "datadate" in df.
 ↪columns else ["firm_id","fyear"])
    df = df.drop_duplicates(subset=["firm_id","fyear"], keep="last")
post_n = len(df)

# Enforce sort order for lag/lead safety
df = df.sort_values(["firm_id","fyear"]).reset_index(drop=True)

# Integrity checks
assert df[["firm_id","fyear"]].isna().sum().sum() == 0, "Missing firm_id or␣
 ↪fyear after typing."
assert df.duplicated(subset=["firm_id","fyear"]).sum() == 0, "Duplicate␣
 ↪firm-year keys remain after dedup."

print(f"Rows: {pre_n:,} -> {post_n:,} after deduplication.")
print("Unique firms:", df["firm_id"].nunique())
print("Year range:", int(df["fyear"].min()), "to", int(df["fyear"].max()))
```

```
Rows: 75,005 -> 75,005 after deduplication.
Unique firms: 11403
Year range: 2014 to 2024
```

1.4.4 2.4 Raw sample composition (no transformations)

[216]:

```
# Minimal sample composition diagnostics (kept lightweight for large panels)

by_year = df.groupby("fyear").agg(
    n_obs=("firm_id","size"),
    n_firms=("firm_id","nunique"),
).reset_index()

display(by_year.tail(12))

# Optional: industry composition if SIC exists
if "sic" in df.columns:
    df["sic2"] = pd.to_numeric(df["sic"], errors="coerce").astype("Int64") //␣
 ↪100
    by_sic2 = df.groupby("sic2").size().sort_values(ascending=False).head(15).
 ↪rename("n_obs").reset_index()
    display(by_sic2)
else:
    print("Note: `sic` not present; skipping industry composition.")
```

	fyear	n_obs	n_firms
0	2014	7455	7455
1	2015	7178	7178
2	2016	6970	6970
3	2017	6831	6831
4	2018	6672	6672
5	2019	6649	6649
6	2020	6703	6703
7	2021	6851	6851
8	2022	6848	6848
9	2023	6611	6611
10	2024	6237	6237

```
Note: `sic` not present; skipping industry composition.
```

1.5 3. Data Cleaning & Missingness Handling (leakage-aware)

1.5.1 3.1 Non-imputable identifiers and label-year setup

We drop observations missing non-imputable identifiers (firm, year).
We also define `label_year = fyear + 1` as the *outcome year* used for forecasting splits.

[217]:

```
# Drop rows with missing key identifiers (already asserted, but keep explicit)
df = df.dropna(subset=["firm_id","fyear"]).copy()
```

```
# label_year defines the year of the t+1 distress label
df["label_year"] = (df["fyear"] + 1).astype("Int64")

# Split masks (defined early; used for leakage-safe preprocessing throughout)
train_pool_mask = df["label_year"] <= CONFIG["TRAIN_CUTOFF_LABEL_YEAR"]
train_pool_years = sorted(df.loc[train_pool_mask, "label_year"].dropna().
 ↪unique().tolist())
if len(train_pool_years) < (CONFIG["VAL_YEARS"] + 1):
    raise ValueError("Not enough label years in train pool to allocate␣
 ↪validation years. Adjust TRAIN_CUTOFF_LABEL_YEAR or VAL_YEARS.")

val_years = train_pool_years[-CONFIG["VAL_YEARS"]:]
val_mask = df["label_year"].isin(val_years)
train_mask = train_pool_mask & (~val_mask)
test_mask = df["label_year"] > CONFIG["TRAIN_CUTOFF_LABEL_YEAR"]

df["split"] = np.where(test_mask, "test", np.where(val_mask, "val", "train"))

print("Split counts:")
display(df["split"].value_counts(dropna=False).to_frame("n_obs"))
print("Validation label_year(s):", val_years)
```

```
Split counts:

       n_obs
split
train  48458
test   19696
val     6851

Validation label_year(s): [2022]
```

1.5.2 3.2 Missingness audit before intervention

[218]:
```
# Identify numeric columns eligible for imputation (exclude identifiers)
id_cols = {"gvkey","firm_id","fyear","label_year","datadate","split"}
numeric_cols = [c for c in df.columns if c not in id_cols and pd.api.types.
 ↪is_numeric_dtype(df[c])]

missing_tbl = (df[numeric_cols].isna().mean().sort_values(ascending=False) *␣
 ↪100).rename("missing_%").to_frame()
missing_tbl["n_missing"] = df[numeric_cols].isna().sum().astype(int)
missing_tbl["dtype"] = [str(df[c].dtype) for c in missing_tbl.index]

display(missing_tbl.head(25))
```

```
        missing_%  n_missing    dtype
dlcch   44.187721      33143  float64
```

```
apalch  40.491967      30371  float64
txach   30.385974      22791  float64
ivstch  25.590294      19194  float64
recch   16.784214      12589  float64
mkvalt  16.465569      12350  float64
sppe    16.317579      12239  float64
act     14.293714      10721  float64
lct     14.259049      10695  float64
xint    14.047064      10536  float64
txditc  12.091194       9069  float64
txp     10.588627       7942  float64
esubc    9.063396       6798  float64
lco      9.038064       6779  float64
aco      9.036731       6778  float64
invch    7.086194       5315  float64
sppiv    5.288981       3967  float64
ivaeq    4.730351       3548  float64
prstkc   4.546364       3410  float64
caps     4.098393       3074  float64
aqc      3.749083       2812  float64
dp       3.629091       2722  float64
re       3.461103       2596  float64
ivch     3.386441       2540  float64
ivao     3.371775       2529  float64
```

1.5.3 3.3 Create missingness indicators (informative signals)

[219]:
```
# Choose a focused set of inputs used for core ratios/events.
REQUIRED_RAW = [
    "at","dlc","dltt","seq","mibt",
    "oibdp","oancf","xint",
    "act","lct","che","rect","invt",
    # dividend-related (we will auto-detect among these later)
    "dv","dvc","dvt","dvp",
]
available_required = [c for c in REQUIRED_RAW if c in df.columns]

# Hard requirement for the distress proxy; fail if absent (unless synthetic␣
↪mode)
HARD_REQUIRED = ["at","dlc","dltt","seq","oibdp"]
missing_hard = [c for c in HARD_REQUIRED if c not in df.columns]
if missing_hard and not USING_SYNTHETIC_DATA:
    raise ValueError(f"Missing required columns for distress proxy construction:
↪ {missing_hard}")

for c in available_required:
    df[f"fmiss_{c}"] = df[c].isna().astype("Int8")
```

```
print("Created missingness flags for:", available_required)
```

```
Created missingness flags for: ['at', 'dlc', 'dltt', 'seq', 'mibt', 'oibdp',
'oancf', 'xint', 'act', 'lct', 'che', 'rect', 'invt', 'dv', 'dvc', 'dvt', 'dvp']
```

1.5.4 3.4 Training-derived size deciles (used for peer imputation groups)

[220]:

```
# Size is based on log(assets) from TRAIN only, to avoid leakage.
at_train = pd.to_numeric(df.loc[train_mask, "at"], errors="coerce")
log_at_train = np.log(at_train.where(at_train > 0)).dropna()

if len(log_at_train) < 50:
    print("WARNING: too few non-missing training `at` values for stable size␣
↪deciles. Using a single size bin.")
    df["size_decile"] = 5  # arbitrary mid-bin
    size_edges = None
else:
    # Use quantile cutpoints computed on training only
    qs = np.linspace(0, 1, 11)
    size_edges = log_at_train.quantile(qs).values
    size_edges[0] = -np.inf
    size_edges[-1] = np.inf

    log_at_all = np.log(pd.to_numeric(df["at"], errors="coerce").where(lambda s:
↪ s > 0))
    df["size_decile"] = pd.cut(log_at_all, bins=size_edges, labels=False,␣
↪include_lowest=True).astype("Float64")

# Fill NA size_decile with training median decile for downstream stability
sd_med = float(pd.to_numeric(df.loc[train_mask, "size_decile"],␣
↪errors="coerce").median())
df["size_decile"] = pd.to_numeric(df["size_decile"], errors="coerce").
↪fillna(sd_med).astype(int)

print("Size decile distribution (train):")
display(df.loc[train_mask, "size_decile"].value_counts().sort_index().
↪to_frame("n_obs"))
```

```
Size decile distribution (train):
```

size_decile	n_obs
0	4824
1	4823
2	4824
3	4823
4	5047

```
5              4823
6              4823
7              4824
8              4823
9              4824
```

1.5.5 3.5 Train-only peer-median imputation (fyear × size_decile)

[221]:
```
# Select peer-imputation columns: economic magnitudes used in ratios / events
# (Keep the scope narrow and explicit to avoid unintended transformations.)
peer_impute_candidates = [
    "at","dlc","dltt","seq","mibt",
    "oibdp","oancf","xint",
    "act","lct","che","rect","invt",
    # optional related cashflow / balance-sheet inputs (used if available)
    "ib","dp","txp","lt","ppent","intan",
]
peer_impute_cols = [c for c in peer_impute_candidates if c in df.columns]

group_cols = ["fyear","size_decile"]

def peer_median_impute(df_in: pd.DataFrame, cols: list[str], train_mask: pd.
 ↪Series, group_cols: list[str]) -> tuple[pd.DataFrame, pd.DataFrame]:
    """Impute NaNs using TRAIN-only medians by group_cols, with TRAIN␣
 ↪(size_decile) then global median fallback."""
    df_out = df_in.copy()
    train = df_out.loc[train_mask, group_cols + cols].copy()
    group_meds = train.groupby(group_cols)[cols].median()
    global_meds = train[cols].median()

    # Intermediate fallback for unseen (fyear, size_decile): use TRAIN␣
 ↪size_decile medians
    size_meds = train.groupby(["size_decile"])[cols].median()
    tmp_size = df_out[["size_decile"]].merge(size_meds.reset_index(),␣
 ↪on="size_decile", how="left")

    # Join group medians (wide) to all rows
    tmp = df_out[group_cols].merge(group_meds.reset_index(), on=group_cols,␣
 ↪how="left", suffixes=("", "_peer"))
    # tmp currently contains the group median columns with original names
    for c in cols:
        peer_med = tmp[c]
        df_out[c] = df_out[c].where(df_out[c].notna(), peer_med)
        size_med = tmp_size[c]
        df_out[c] = df_out[c].where(df_out[c].notna(), size_med)
        df_out[c] = df_out[c].where(df_out[c].notna(), global_meds[c])
    impute = pd.DataFrame({
```

```
        "col": cols,
        "n_imputed": [int(df_in[c].isna().sum() - df_out[c].isna().sum()) for c␣
↪in cols],
        "train_global_median": [float(global_meds[c]) if pd.
↪notna(global_meds[c]) else np.nan for c in cols],
    })
    return df_out, impact

df_pre_impute_snapshot = df.copy(deep=True)
df, peer_impact = peer_median_impute(df, peer_impute_cols, train_mask,␣
↪group_cols)

display(peer_impact.sort_values("n_imputed", ascending=False).head(15))
```

	col	n_imputed	train_global_median
8	act	10721	20430.00
9	lct	10695	10099.00
7	xint	10536	418.00
15	txp	7942	0.00
14	dp	2722	1036.00
5	oibdp	2142	957.35
17	ppent	1723	8290.00
11	rect	880	5764.00
4	mibt	842	0.00
18	intan	654	1186.00
12	invt	652	291.05
6	oancf	303	788.70
2	dltt	188	2174.50
16	lt	101	34749.00
1	dlc	31	304.00

1.5.6 3.6 KNN imputation on core balance-sheet items (train-fit; signed-log transform)

[222]:
```
knn_cols = [c for c in␣
↪["at","act","lct","che","rect","invt","dlc","dltt","seq","ppent","intan"] if␣
↪c in df.columns]

if len(knn_cols) >= 3:
    Z = df[knn_cols + ["fyear","size_decile"]].copy()
    # Transform magnitudes for distance stability
    for c in knn_cols:
        Z[c] = signed_log1p(Z[c])
    # Keep fyear/size_decile in levels
    Z["fyear"] = pd.to_numeric(Z["fyear"], errors="coerce")
    Z["size_decile"] = pd.to_numeric(Z["size_decile"], errors="coerce")
```

```
    imputer = KNNImputer(n_neighbors=CONFIG["KNN_K"], weights="distance")
    imputer.fit(Z.loc[train_mask, :])

    Z_imp = pd.DataFrame(imputer.transform(Z), columns=Z.columns, index=Z.index)

    # Invert signed-log transform back for magnitudes
    for c in knn_cols:
        # inverse of signed_log1p: sign(z)*(exp(|z|)-1)
        z = pd.to_numeric(Z_imp[c], errors="coerce")
        df[c] = np.sign(z) * (np.expm1(np.abs(z)))
else:
    print("Skipping KNN imputation: insufficient columns available.")
```

1.5.7 3.7 Guardrail capping of imputed magnitudes (train quantile bands)

[223]:
```
cap_cols = peer_impute_cols  # apply to columns we actively imputed
bounds = {}

for c in cap_cols:
    lo, hi = winsorize_train_bounds(df_pre_impute_snapshot.loc[train_mask, c],␣
 ↪CONFIG["IMPUTE_LO_Q"], CONFIG["IMPUTE_HI_Q"])
    bounds[c] = {"lo": lo, "hi": hi}
    df[c] = apply_bounds(df[c], lo, hi)

bounds_df = pd.DataFrame({c: (v["lo"], v["hi"]) for c,v in bounds.items()},␣
 ↪index=["lo","hi"]).T
bounds_df.index.name = "col"
display(bounds_df.head(15))
```

col	lo	hi
at	0.93	50746128.37
dlc	0.00	3229014.45
dltt	0.00	11798057.82
seq	-155487.96	14903048.78
mibt	-1326.96	963289.56
oibdp	-157336.02	4359988.88
oancf	-136407.56	3365799.80
xint	0.00	464656.90
act	0.08	10216167.52
lct	0.47	8445475.37
che	0.00	6054868.56
rect	0.00	16780556.04
invt	0.00	2736334.87
ib	-459285.27	1785532.79
dp	0.00	1363783.00

1.5.8 3.8 Imputation impact audit (pre vs post)

[224]:
```python
audit_cols = [c for c in ["at","dlc","dltt","seq","oibdp","oancf","act","lct"]␣
↪if c in df.columns]

def dist_summary(x: pd.Series) -> dict:
    x = pd.to_numeric(x, errors="coerce")
    return {
        "n": int(x.notna().sum()),
        "mean": float(x.mean()) if x.notna().any() else np.nan,
        "p50": float(x.median()) if x.notna().any() else np.nan,
        "p10": float(x.quantile(0.10)) if x.notna().any() else np.nan,
        "p90": float(x.quantile(0.90)) if x.notna().any() else np.nan,
    }

rows = []
for c in audit_cols:
    pre = dist_summary(df_pre_impute_snapshot[c])
    post = dist_summary(df[c])
    rows.append({
        "col": c,
        "n_pre": pre["n"],
        "n_post": post["n"],
        "mean_pre": pre["mean"],
        "mean_post": post["mean"],
        "p50_pre": pre["p50"],
        "p50_post": post["p50"],
    })
impact_tbl = pd.DataFrame(rows).sort_values("col")
display(impact_tbl)
```

	col	n_pre	n_post	mean_pre	mean_post	p50_pre	p50_post
6	act	64284	75005	6.440956e+05	4.876191e+05	23074.0	38683.0
0	at	75005	75005	5.597314e+06	2.060420e+06	96379.0	96379.0
1	dlc	74974	75005	3.666371e+05	8.767585e+04	588.0	589.0
2	dltt	74817	75005	9.226323e+05	4.595095e+05	3416.0	3420.7
7	lct	64310	75005	4.755088e+05	3.211051e+05	11405.0	18694.0
5	oancf	74702	75005	1.976674e+05	1.198831e+05	644.0	638.0
4	oibdp	72863	75005	2.472644e+05	1.546921e+05	800.4	967.0
3	seq	75002	75005	9.623007e+05	6.320920e+05	34787.5	34786.0

1.6 4. Exploratory Data Analysis (EDA)

EDA focuses on **signal strength and data quality**, not exhaustive plotting.
At this stage we describe the imputed-but-not-modeled input space, by split.

1.6.1 4.1 Summary statistics by split (key magnitudes)

[225]:
```python
eda_cols = [c for c in ["at","dlc","dltt","seq","oibdp","oancf","xint"] if c in df.columns]

def split_describe(df_in: pd.DataFrame, cols: list[str]) -> pd.DataFrame:
    out = []
    for sp in ["train","val","test"]:
        d = df_in.loc[df_in["split"]==sp, cols].describe(percentiles=[0.01,0.1,0.5,0.9,0.99]).T
        d.insert(0, "split", sp)
        d.insert(1, "col", d.index)
        out.append(d.reset_index(drop=True))
    return pd.concat(out, ignore_index=True)

desc_tbl = split_describe(df, eda_cols)
display(desc_tbl.head(20))
```

```
    split     col    count          mean           std        min          1%  \
0   train      at  48458.0  1.958101e+06  6.596407e+06       0.93      0.9300
1   train     dlc  48458.0  8.522942e+04  3.910776e+05       0.00      0.0000
2   train    dltt  48458.0  4.456461e+05  1.581484e+06       0.00      0.0000
3   train     seq  48458.0  5.974454e+05  1.953979e+06 -155487.96 -155258.2728
4   train   oibdp  48458.0  1.475527e+05  5.549821e+05 -157336.02 -152927.2400
5   train   oancf  48458.0  1.158057e+05  4.347518e+05 -136407.56 -135978.6000
6   train    xint  48458.0  2.317919e+04  6.383769e+04       0.00      0.0000
7     val      at   6851.0  2.165063e+06  6.845755e+06       0.93      1.0000
8     val     dlc   6851.0  8.535066e+04  3.905082e+05       0.00      0.0000
9     val    dltt   6851.0  4.996244e+05  1.690123e+06       0.00      0.0000
10    val     seq   6851.0  7.102562e+05  2.141541e+06 -155487.96 -126339.0000
11    val   oibdp   6851.0  1.644824e+05  6.147170e+05 -157336.02 -157336.0200
12    val   oancf   6851.0  1.220935e+05  4.707193e+05 -136407.56 -136407.5600
13    val    xint   6851.0  2.226171e+04  6.119281e+04       0.00      0.0000
14   test      at  19696.0  2.275757e+06  7.193903e+06       0.93      2.9685
15   test     dlc  19696.0  9.450359e+04  3.961111e+05       0.00      0.0000
16   test    dltt  19696.0  4.796641e+05  1.637741e+06       0.00      0.0000
17   test     seq  19696.0  6.901446e+05  2.146861e+06 -155487.96 -155487.9600
18   test   oibdp  19696.0  1.688519e+05  6.187964e+05 -157336.02 -157336.0200
19   test   oancf  19696.0  1.291457e+05  4.742978e+05 -136407.56 -136407.5600

          10%        50%        90%           99%          max
0     534.985  90113.000  3790678.7  5.067805e+07  50746128.37
1       0.000    304.700    92567.9  3.228734e+06   3229014.45
2       0.000   2174.000   920901.9  1.178757e+07  11798057.82
3    -382.300  33096.000  1252878.4  1.487620e+07  14903048.78
4  -12153.400   1133.300   319682.8  4.210592e+06   4359988.88
5  -10121.600    781.000   239939.4  3.324934e+06   3365799.80
6       0.000    628.000    69636.6  4.195421e+05    464656.90
```

7	918.000	125582.000	4515726.0	5.049778e+07	50746128.37
8	0.000	1026.000	90787.0	3.164053e+06	3229014.45
9	0.000	6173.000	1075558.0	1.179806e+07	11798057.82
10	-2.380	55807.000	1534804.0	1.490305e+07	14903048.78
11	-31901.000	627.000	382208.0	4.359989e+06	4359988.88
12	-29644.000	327.380	257984.0	3.365800e+06	3365799.80
13	0.000	632.000	66334.0	3.829785e+05	464656.90
14	1162.300	103152.500	4657672.5	5.074613e+07	50746128.37
15	0.000	1515.000	125309.0	2.962161e+06	3229014.45
16	0.000	6473.100	1035513.0	1.179806e+07	11798057.82
17	-363.000	33184.000	1484313.0	1.490305e+07	14903048.78
18	-31389.000	747.000	403014.0	4.359989e+06	4359988.88
19	-25528.500	467.295	286839.0	3.365800e+06	3365799.80

1.6.2 4.2 Missingness rates by split (key inputs)

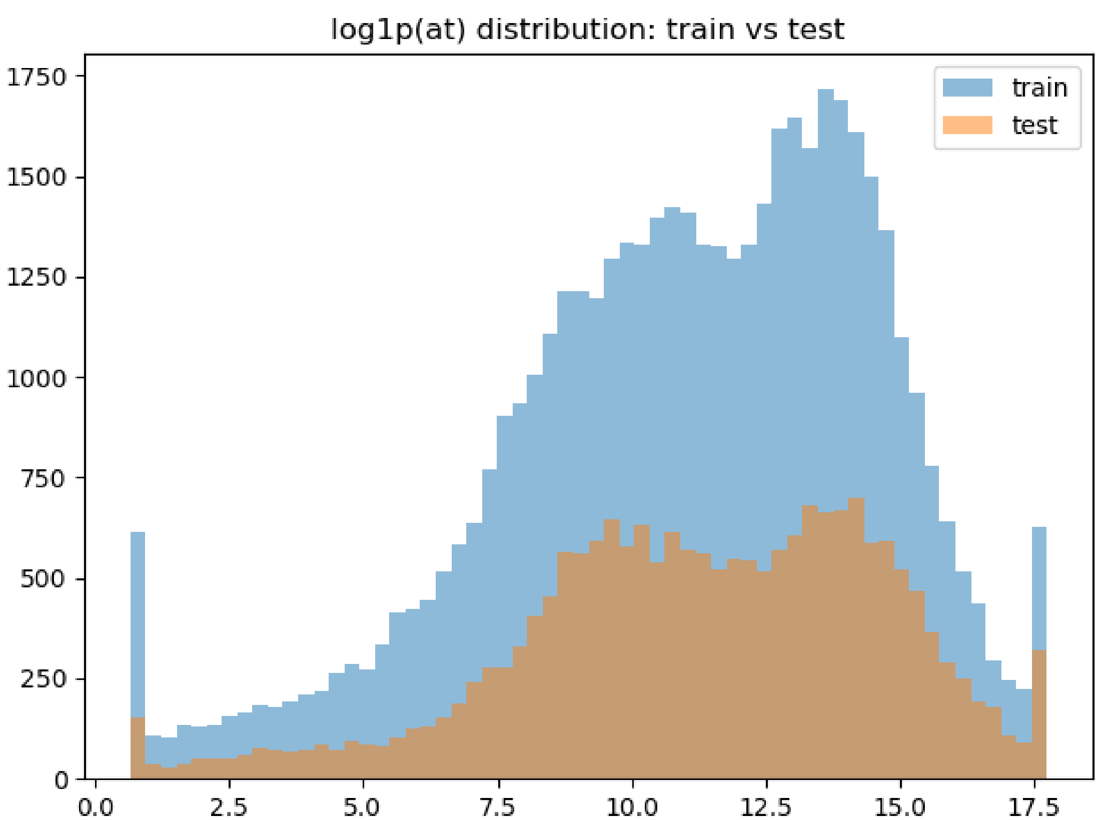
[226]:
```
miss_cols = [c for c in available_required if f"fmiss_{c}" in df.columns]
miss_by_split = (
    df.groupby("split")[ [f"fmiss_{c}" for c in available_required if f"fmiss_{c}" in df.columns] ]
      .mean()
      .T
)
miss_by_split.index = [i.replace("fmiss_","") for i in miss_by_split.index]
miss_by_split = (miss_by_split * 100).round(2)
display(miss_by_split)
```

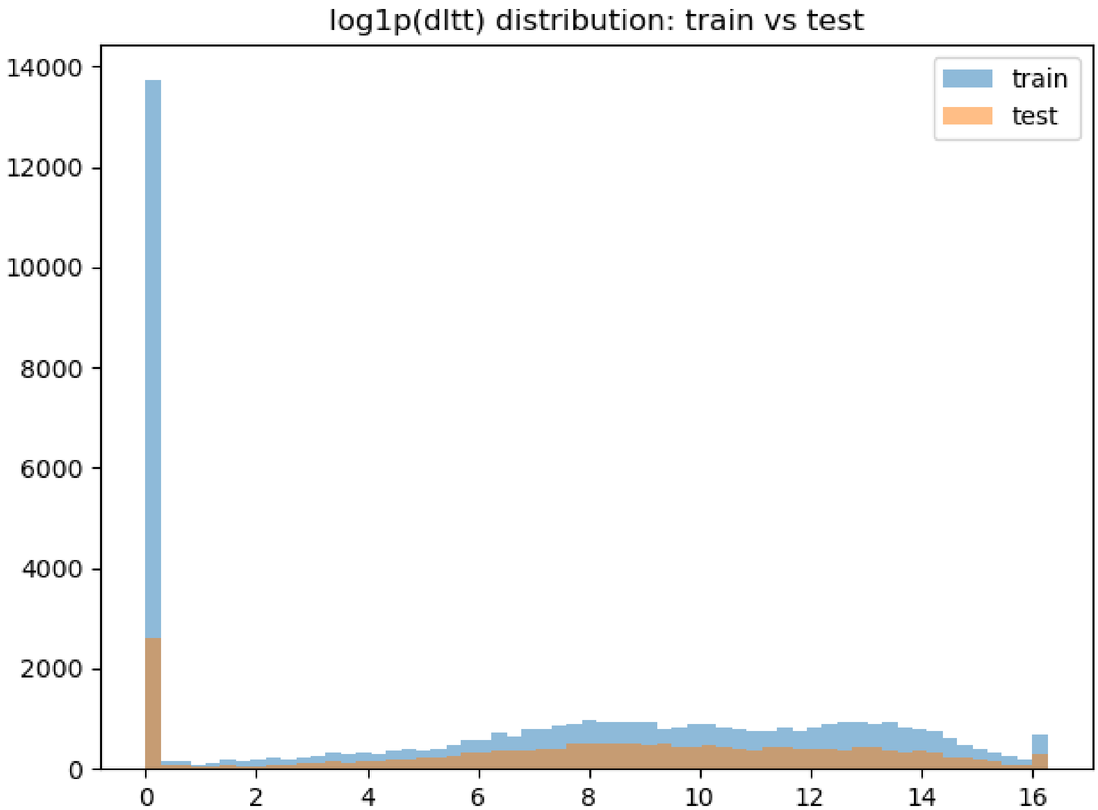
split	test	train	val
at	0.0	0.0	0.0
dlc	0.03	0.05	0.03
dltt	0.16	0.29	0.23
seq	0.0	0.01	0.0
mibt	0.76	1.3	0.89
oibdp	2.65	2.94	2.89
oancf	0.28	0.46	0.35
xint	14.57	13.92	13.43
act	13.62	14.66	13.66
lct	13.62	14.6	13.69
che	0.0	0.01	0.0
rect	1.26	1.12	1.26
invt	0.96	0.83	0.9
dv	1.36	1.59	1.45
dvc	0.55	1.14	0.66
dvt	0.55	1.14	0.66
dvp	0.05	0.05	0.04

1.6.3 4.3 Visual sanity-check plots (train vs test distributions)

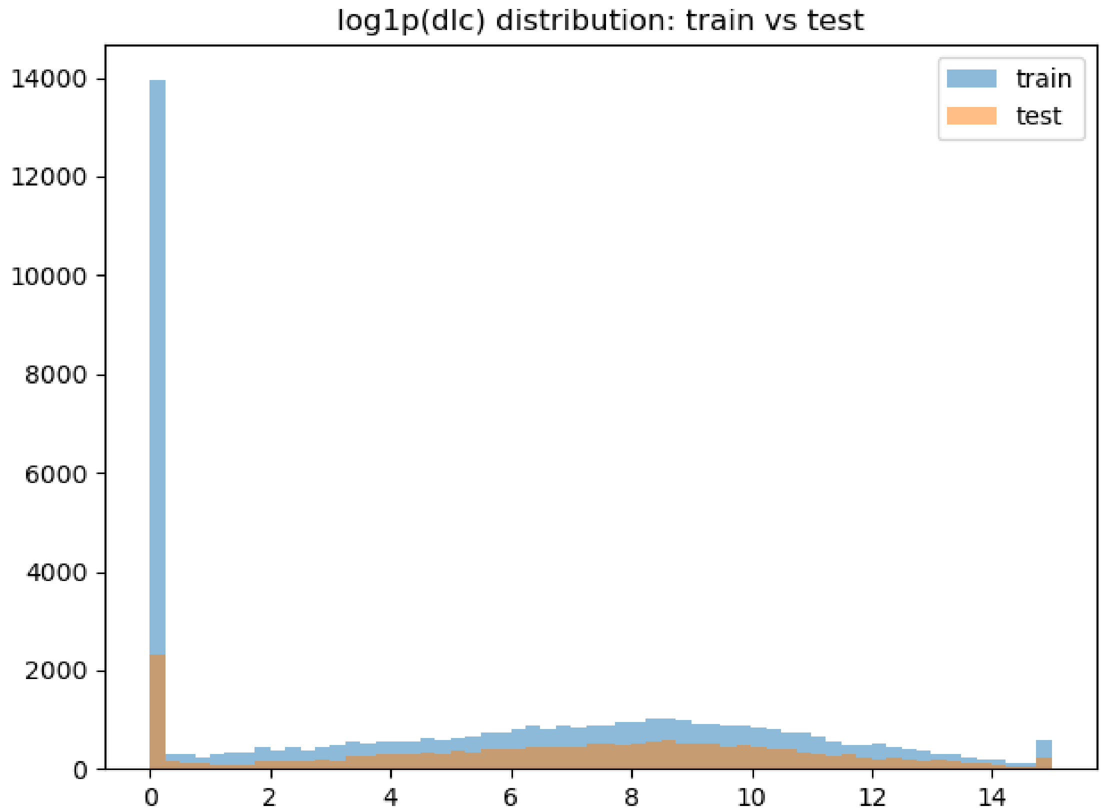
[227]:

```python
# Lightweight plots to spot gross drift / outliers.
plot_cols = [c for c in ["at","dltt","dlc","oibdp","oancf"] if c in df.columns]

for c in plot_cols[:3]:
    a = pd.to_numeric(df.loc[df["split"]=="train", c], errors="coerce")
    b = pd.to_numeric(df.loc[df["split"]=="test", c], errors="coerce")
    plt.figure()
    plt.hist(np.log1p(a.dropna()), bins=60, alpha=0.5, label="train")
    plt.hist(np.log1p(b.dropna()), bins=60, alpha=0.5, label="test")
    plt.title(f"log1p({c}) distribution: train vs test")
    plt.legend()
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / f"eda_log1p_{c}_train_vs_test.png",␣
 ↪dpi=140)
    plt.show()
```





log1p(dltt) distribution: train vs test
train
test
14000
12000
10000
8000
6000
4000
2000
0
0
2
4
6
8
10
12
14
16



1.7 5. Feature Engineering & Target Construction

This section constructs **all derived features explicitly** from Compustat-style raw items, including: - debt aggregates and leverage ratios, - cash-flow-to-debt ratios, - log size and log market value, - the NA-aware distress proxy and the next-year label.

Design choice: ratios with non-positive denominators are treated as **extreme tail states** (encoded via `+∞` then converted to `NaN` before modeling), rather than silently set to zero.

1.7.1 5.1 Debt, capital, and operating aggregates

[228]:
```
# Debt aggregate
df["total_debt"] = df[["dlc","dltt"]].sum(axis=1, min_count=1)

# Equity plus minority interest (if available)
if "mibt" in df.columns:
    df["equity_plus_mi"] = df[["seq","mibt"]].sum(axis=1, min_count=1)
else:
    df["equity_plus_mi"] = df["seq"]

# Total capital and a non-positive capital flag
```

```
df["total_capital"] = df[["total_debt","equity_plus_mi"]].sum(axis=1,␣
↪min_count=1)
df["cap_nonpos_flag"] = (pd.to_numeric(df["total_capital"], errors="coerce") <=␣
↪0).astype("Int8")

# EBITDA proxy
df["ebitda_proxy"] = df["oibdp"] if "oibdp" in df.columns else np.nan
df["ebitda_nonpos_flag"] = (pd.to_numeric(df["ebitda_proxy"], errors="coerce")␣
↪<= 0).astype("Int8")

# Log transforms (explicit, auditable)
df["log_at"] = np.log(pd.to_numeric(df["at"], errors="coerce").where(lambda s:␣
↪s > 0))
```

1.7.2 5.2 Leverage, coverage, and cash-flow ratios (explicit NA / edge-case rules)

[229]:
```
# --- Leverage ratios ---
debt = pd.to_numeric(df["total_debt"], errors="coerce")
cap = pd.to_numeric(df["total_capital"], errors="coerce")
ebitda = pd.to_numeric(df["ebitda_proxy"], errors="coerce")

# debt-to-capital: if capital <= 0 treat as extreme leverage (+inf)
df["sp_debt_to_capital"] = debt / cap
df.loc[cap <= 0, "sp_debt_to_capital"] = np.inf

# debt-to-EBITDA: if EBITDA <= 0 treat as extreme leverage (+inf)
df["sp_debt_to_ebitda"] = debt / ebitda
df.loc[ebitda <= 0, "sp_debt_to_ebitda"] = np.inf

# --- Cash flow proxies and ratios ---
# CFO proxy: oancf (operating cash flow)
cfo = pd.to_numeric(df["oancf"], errors="coerce") if "oancf" in df.columns else␣
↪np.nan

# FFO proxy: a pragmatic approximation in Compustat annual data
# (Add back interest; optionally adjust for taxes paid if available.)
xint = pd.to_numeric(df["xint"], errors="coerce") if "xint" in df.columns else␣
↪np.nan
txp = pd.to_numeric(df["txp"], errors="coerce") if "txp" in df.columns else np.
↪nan
df["ffo_proxy"] = cfo + xint
if "txp" in df.columns:
    # keep conservative: subtract taxes paid if available (makes FFO "cash␣
↪available" smaller)
    df["ffo_proxy"] = df["ffo_proxy"] - txp
```

```
df["sp_cfo_to_debt"] = cfo / debt
df["sp_ffo_to_debt"] = pd.to_numeric(df["ffo_proxy"], errors="coerce") / debt

# Free cash flow proxies (optional; use if capx exists)
if "capx" in df.columns:
    capx = pd.to_numeric(df["capx"], errors="coerce")
    df["focf_proxy"] = cfo - capx
    df["sp_focf_to_debt"] = df["focf_proxy"] / debt
else:
    df["focf_proxy"] = np.nan
    df["sp_focf_to_debt"] = np.nan

# Debt service capacity proxy (DCF): CFO + interest expense (simple)
df["dcf_proxy"] = cfo + xint
df["sp_dcf_to_debt"] = pd.to_numeric(df["dcf_proxy"], errors="coerce") / debt

# --- Interest coverage (computed, but typically excluded from MODEL_FEATS to␣
↪prevent circularity) ---
if "xint" in df.columns and "oibdp" in df.columns:
    denom_floor = 1e-3
    raw_cov = safe_divide(pd.to_numeric(df["oibdp"], errors="coerce"), pd.
↪to_numeric(df["xint"], errors="coerce"), denom_floor=denom_floor)
    # cap very large values for numeric stability
    cov_cap = 50.0
    df["sp_interest_coverage"] = raw_cov.clip(lower=-cov_cap, upper=cov_cap)
    df["cov_capped_flag"] = (raw_cov.abs() > cov_cap).astype("Int8")
else:
    df["sp_interest_coverage"] = np.nan
    df["cov_capped_flag"] = 0

# Replace any remaining +/-inf in ratio fields with np.inf explicitly (will be␣
↪handled later)
ratio_cols =␣
↪["sp_debt_to_capital","sp_debt_to_ebitda","sp_cfo_to_debt","sp_ffo_to_debt","sp_focf_to_deb
for c in ratio_cols:
    if c in df.columns:
        df[c] = df[c].replace([np.inf, -np.inf], [np.inf, -np.inf])
```

1.7.3 5.3 Distress proxy (fiscal year t) and next-year supervised label (t+1)

[230]:
```
# Distress proxy thresholds (frozen and documented)
DISTRESS_RULE = {
    "FFO_TO_DEBT_LT": 0.15,
    "DEBT_TO_CAPITAL_GT": 0.55,
    "DEBT_TO_EBITDA_GT": 4.5,
    "NEG_EQUITY_SEQ_LE": 0.0,
}
```

```
ffo_to_debt = ensure_nullable_float(df["sp_ffo_to_debt"])
debt_to_cap = ensure_nullable_float(df["sp_debt_to_capital"])
debt_to_ebitda = ensure_nullable_float(df["sp_debt_to_ebitda"])
seq = ensure_nullable_float(df["seq"])

cond_ffo = ffo_to_debt < DISTRESS_RULE["FFO_TO_DEBT_LT"]
cond_cap = debt_to_cap > DISTRESS_RULE["DEBT_TO_CAPITAL_GT"]
cond_ebitda = debt_to_ebitda > DISTRESS_RULE["DEBT_TO_EBITDA_GT"]

high_leverage = cond_ffo & cond_cap & cond_ebitda
neg_equity = seq <= DISTRESS_RULE["NEG_EQUITY_SEQ_LE"]

distress_proxy = high_leverage | neg_equity
df["distress_proxy_t"] = distress_proxy  # BooleanDtype with <NA>
df["distress_dummy_t"] = df["distress_proxy_t"].astype("Int8")  # {0,1,<NA>}

# Next-year target: lead of distress_dummy_t within firm
df["target_next_year_distress"] = df.groupby("firm_id")["distress_dummy_t"].
↪shift(-1).astype("Int8")

# Label availability / attrition
df["has_next_year_obs"] = df.groupby("firm_id")["fyear"].shift(-1).notna().
↪astype("Int8")

print("Distress prevalence (by split) - based on next-year target:")
prev = df.groupby("split")["target_next_year_distress"].mean().
↪rename("target_rate").to_frame()
display(prev)

print("Share of observations with next-year observation (attrition diagnostic):
↪")
display(df.groupby("split")["has_next_year_obs"].mean().rename("has_next_rate").
↪to_frame())
```

```
Distress prevalence (by split) - based on next-year target:

       target_rate
split
test       0.21751
train     0.209499
val       0.217147

Share of observations with next-year observation (attrition diagnostic):

       has_next_rate
split
test        0.629773
train       0.924161
```

```
val          0.93636
```

1.7.4 5.4 Target prevalence and attrition diagnostics (by year and size)

[231]:
```
# Target prevalence by label year
by_label_year = df.groupby(["label_year","split"]).agg(
    n_obs=("firm_id","size"),
    target_rate=("target_next_year_distress","mean"),
    has_next_rate=("has_next_year_obs","mean"),
).reset_index()

display(by_label_year.tail(15))

# By size decile (train pool), to assess composition effects
by_size = df.groupby(["size_decile","split"]).agg(
    n_obs=("firm_id","size"),
    target_rate=("target_next_year_distress","mean"),
).reset_index()

display(by_size.sort_values(["split","size_decile"]).head(30))
```

	label_year	split	n_obs	target_rate	has_next_rate
0	2015	train	7455	0.215562	0.908518
1	2016	train	7178	0.20624	0.915297
2	2017	train	6970	0.204913	0.922812
3	2018	train	6831	0.201199	0.927683
4	2019	train	6672	0.226146	0.92521
5	2020	train	6649	0.222044	0.937434
6	2021	train	6703	0.190583	0.934656
7	2022	val	6851	0.217147	0.93636
8	2023	test	6848	0.217639	0.923919
9	2024	test	6611	0.217377	0.919226
10	2025	test	6237	<NA>	0.0

	size_decile	split	n_obs	target_rate
0	0	test	1500	0.479467
3	1	test	1671	0.288297
6	2	test	2182	0.237537
9	3	test	2232	0.221341
12	4	test	2045	0.239784
15	5	test	1928	0.189456
18	6	test	1823	0.174103
21	7	test	1925	0.169643
24	8	test	2072	0.152852
27	9	test	2318	0.121333
1	0	train	4824	0.457619
4	1	train	4823	0.315634
7	2	train	4824	0.222716

```
10             3  train   4823      0.172815
13             4  train   5047      0.202889
16             5  train   4823      0.162199
19             6  train   4823      0.133273
22             7  train   4824      0.146758
25             8  train   4823      0.159638
28             9  train   4824      0.133462
2              0    val    555      0.476096
5              1    val    600      0.291139
8              2    val    649      0.252443
11             3    val    652      0.181965
14             4    val    763       0.23488
17             5    val    735      0.181149
20             6    val    717      0.177914
23             7    val    676      0.168731
26             8    val    748      0.173239
29             9    val    756       0.11967
```

1.7.5 5.5 Event indicators (evt_*) for decision support

Events are discrete, interpretable signals designed for operational triage.
They are calibrated **using training data only** (when thresholds are estimated), and we explicitly **exclude** events mechanically tied to the distress-definition ratios (leverage/coverage) from the predictive feature set.

Events implemented here (subject to data availability): - Dividend cut / suspension / initiation - Liquidity squeeze (current ratio < 1.0) and quick-ratio squeeze (< 0.8) - EBITDA drop (vs. t-1) and CFO drop (vs. t-1)

[232]:
```
# Ensure sorting already enforced
assert df.index.is_monotonic_increasing

# Lag helpers
def lag(df_in: pd.DataFrame, col: str, n: int = 1) -> pd.Series:
    return df_in.groupby("firm_id")[col].shift(n)

# Identify dividend column (prefer dvc if present; else dv / dvt / dvp)
dividend_candidates = ["dvc","dv","dvt","dvp"]
div_col = next((c for c in dividend_candidates if c in df.columns), None)

if div_col is None:
    print("Dividend column not found (looked for dvc/dv/dvt/dvp). Dividend␣
 ↪events will be NaN.")
    df["evt_divcut"] = np.nan
    df["evt_divsusp"] = np.nan
    df["evt_divinit"] = np.nan
else:
    # Use absolute value (guard against sign conventions)
```

```
    df["dv_obs"] = pd.to_numeric(df[div_col], errors="coerce").abs()
    df["dv_obs_l1"] = lag(df, "dv_obs", 1)

# Liquidity ratios
if "act" in df.columns and "lct" in df.columns:
    df["current_ratio"] = safe_divide(df["act"], df["lct"], denom_floor=1e-6)
else:
    df["current_ratio"] = np.nan

if "act" in df.columns and "lct" in df.columns:
    if "invt" in df.columns:
        df["quick_ratio"] = safe_divide(pd.to_numeric(df["act"],␣
↪errors="coerce") - pd.to_numeric(df["invt"], errors="coerce"),
                                        df["lct"], denom_floor=1e-6)
    elif "che" in df.columns and "rect" in df.columns:
        df["quick_ratio"] = safe_divide(pd.to_numeric(df["che"],␣
↪errors="coerce") + pd.to_numeric(df["rect"], errors="coerce"),
                                        df["lct"], denom_floor=1e-6)
    else:
        df["quick_ratio"] = df["current_ratio"]
else:
    df["quick_ratio"] = np.nan

# EBITDA and CFO lags for deterioration events
if "ebitda_proxy" in df.columns:
    df["ebitda_l1"] = lag(df, "ebitda_proxy", 1)
if "oancf" in df.columns:
    df["cfo_l1"] = lag(df, "oancf", 1)
```

5.5.1 Dividend policy events (training-calibrated cut threshold)

[233]:
```
event_params = {}

if div_col is None:
    pass
else:
    # YoY % change among observed payers with meaningful baseline
    dv_l1 = pd.to_numeric(df["dv_obs_l1"], errors="coerce")
    dv = pd.to_numeric(df["dv_obs"], errors="coerce")
    df["div_pct_change"] = np.where(dv_l1 > 1e-2, (dv - dv_l1) / dv_l1, np.nan)

    payer_train = train_mask & (dv_l1 > 0) & pd.notna(df["div_pct_change"])
    if payer_train.sum() >= 50:
        cut_thr = float(np.nanpercentile(df.loc[payer_train, "div_pct_change"],␣
↪10))
    else:
        cut_thr = -0.25
```

```
    # Bound cut threshold to avoid pathological values
    cut_thr = float(np.clip(cut_thr, -0.50, -0.10))
    event_params["DIV_CUT_THR_P10_BOUNDED"] = cut_thr

    # Dividend cut: large negative YoY change among payers
    df["evt_divcut"] = (df["div_pct_change"] <= cut_thr).astype("Int8")

    # Suspension: payer last year, ~zero dividend now
    df["evt_divsusp"] = ((dv_l1 > 0) & (dv.fillna(0) <= 1e-4)).astype("Int8")

    # Initiation: ~zero last year, dividend now positive
    df["evt_divinit"] = ((dv_l1.fillna(0) <= 1e-4) & (dv > 1e-4)).astype("Int8")

    print(f"Dividend cut threshold (train P10 bounded): {cut_thr:.3f}")
   ␣
 ↪display(df[["dv_obs","dv_obs_l1","div_pct_change","evt_divcut","evt_divsusp","evt_divinit"]].
 ↪head(8))
```

Dividend cut threshold (train P10 bounded): -0.500

	dv_obs	dv_obs_l1	div_pct_change	evt_divcut	evt_divsusp	evt_divinit
0	11905.0	NaN	NaN	0	0	1
1	13697.0	11905.0	0.150525	0	0	0
2	15447.0	13697.0	0.127765	0	0	0
3	17287.0	15447.0	0.119117	0	0	0
4	18854.0	17287.0	0.090646	0	0	0
5	20593.0	18854.0	0.092235	0	0	0
6	11218.0	20593.0	-0.455252	0	0	0
7	22179.0	11218.0	0.977090	0	0	0

5.5.2 Liquidity squeeze events

[234]:
```
df["evt_liq_squeeze"] = (pd.to_numeric(df["current_ratio"], errors="coerce") <␣
 ↪1.0).astype("Int8")
df["evt_quick_squeeze"] = (pd.to_numeric(df["quick_ratio"], errors="coerce") <␣
 ↪0.8).astype("Int8")

display(df[["current_ratio","quick_ratio","evt_liq_squeeze","evt_quick_squeeze"]].
 ↪head(8))
```

	current_ratio	quick_ratio	evt_liq_squeeze	evt_quick_squeeze
0	1.857971	0.749443	0	1
1	0.001967	-1.169719	1	1
2	1.813861	0.712973	0	1
3	1.735209	0.593500	0	1
4	1.747436	0.628349	0	1
5	1.775434	0.585567	0	1
6	1.977964	0.836194	0	0

7	1.492971	0.508954	0	1

5.5.3 Operating deterioration events (vs. t-1)

[235]:
```
# EBITDA drop: requires lagged EBITDA observed and positive
if "ebitda_proxy" in df.columns:
    e = pd.to_numeric(df["ebitda_proxy"], errors="coerce")
    e_l1 = pd.to_numeric(df["ebitda_l1"], errors="coerce")
    ratio = e / e_l1
    evt = ((e_l1 > 0) & ((ratio < 0.5) | (e <= 0))).astype("Int8")
    evt = evt.where(pd.notna(e_l1), other=pd.NA).astype("Int8")
    df["evt_ebitdadrop"] = evt
else:
    df["evt_ebitdadrop"] = pd.NA

# CFO drop: requires lagged CFO observed and positive
if "oancf" in df.columns:
    c = pd.to_numeric(df["oancf"], errors="coerce")
    c_l1 = pd.to_numeric(df["cfo_l1"], errors="coerce")
    ratio = c / c_l1
    evt = ((c_l1 > 0) & ((ratio < 0.5) | (c <= 0))).astype("Int8")
    evt = evt.where(pd.notna(c_l1), other=pd.NA).astype("Int8")
    df["evt_cfdrop"] = evt
else:
    df["evt_cfdrop"] = pd.NA

display(df[["ebitda_proxy","ebitda_l1","evt_ebitdadrop","oancf","cfo_l1","evt_cfdrop"]].
↪head(10))
```

	ebitda_proxy	ebitda_l1	evt_ebitdadrop	oancf	cfo_l1	evt_cfdrop
0	113.4	NaN	<NA>	46987.00	NaN	<NA>
1	97361.0	113.4	0	65171.00	46987.0	0
2	121286.0	97361.0	0	97805.00	65171.0	0
3	126988.0	121286.0	0	64617.00	97805.0	0
4	105555.0	126988.0	0	70258.00	64617.0	0
5	122894.0	105555.0	0	76928.00	70258.0	0
6	138308.0	122894.0	0	97896.00	76928.0	0
7	158338.0	138308.0	0	85564.00	97896.0	0
8	141213.0	158338.0	0	-27533.00	85564.0	1
9	124265.0	141213.0	0	144.26	-27533.0	0

5.5.4 Event dictionary (appendix-ready)

[236]:
```
event_dict_rows = [
    {"event":"evt_divcut", "definition":"Dividend YoY % change <= training P10␣
↪threshold (bounded [-0.50,-0.10])", "inputs":div_col or "N/A", "calibration":
↪"train-only"},
    {"event":"evt_divsusp", "definition":"Dividend >0 at t-1 and ~0 at t",␣
↪"inputs":div_col or "N/A", "calibration":"none"},
```

```
    {"event":"evt_divinit", "definition":"Dividend ~0 at t-1 and >0 at t",␣
 ↪"inputs":div_col or "N/A", "calibration":"none"},
    {"event":"evt_liq_squeeze", "definition":"Current ratio < 1.0", "inputs":
 ↪"act,lct", "calibration":"fixed threshold"},
    {"event":"evt_quick_squeeze", "definition":"Quick ratio < 0.8", "inputs":
 ↪"act,lct,invt (or che+rect)", "calibration":"fixed threshold"},
    {"event":"evt_ebitdadrop", "definition":"EBITDA <=0 OR EBITDA/
 ↪EBITDA_{t-1}<0.5 (requires EBITDA_{t-1}>0)", "inputs":"oibdp", "calibration":
 ↪"fixed threshold"},
    {"event":"evt_cfdrop", "definition":"CFO <=0 OR CFO/CFO_{t-1}<0.5 (requires␣
 ↪CFO_{t-1}>0)", "inputs":"oancf", "calibration":"fixed threshold"},
]
event_dict = pd.DataFrame(event_dict_rows)
event_dict["parameter"] = event_dict["event"].map(lambda e: json.dumps({k:v for␣
 ↪k,v in event_params.items()}) if e=="evt_divcut" else "")
display(event_dict)
```

```
               event                                          definition  \
0         evt_divcut  Dividend YoY % change <= training P10 threshol…
1        evt_divsusp                     Dividend >0 at t-1 and ~0 at t
2        evt_divinit                     Dividend ~0 at t-1 and >0 at t
3    evt_liq_squeeze                                Current ratio < 1.0
4  evt_quick_squeeze                                  Quick ratio < 0.8
5     evt_ebitdadrop  EBITDA <=0 OR EBITDA/EBITDA_{t-1}<0.5 (require…
6         evt_cfdrop  CFO <=0 OR CFO/CFO_{t-1}<0.5 (requires CFO_{t-…

                       inputs      calibration  \
0                         dvc       train-only
1                         dvc             none
2                         dvc             none
3                     act,lct  fixed threshold
4  act,lct,invt (or che+rect)  fixed threshold
5                       oibdp  fixed threshold
6                       oancf  fixed threshold

                           parameter
0  {"DIV_CUT_THR_P10_BOUNDED": -0.5}
1
2
3
4
5
6
```

1.8 6. Preprocessing for Modeling (train-only fitting)

Preprocessing design principles: - **Train-only fitting:** imputation (if needed), winsorization bounds, and scaling are all fit on *train* only. - **Binary events stay in levels** (0/1) to preserve interpretability and prevalence. - **Leakage audit:** variables that mechanically define the distress proxy are excluded from MODEL_FEATS.

1.8.1 6.1 Feature set definition and leakage audit

[237]:
```
# Features that participate in the distress proxy definition (must be excluded␣
↪from predictors)
DISTRESS_DEFINITION_VARS = {
    "sp_ffo_to_debt",
    "sp_debt_to_capital",
    "sp_debt_to_ebitda",
    "seq",
}

# Candidate continuous predictors (explicit; adjust as needed)
continuous_feats_raw = [
    "sp_cfo_to_debt",
    "sp_focf_to_debt",
    "sp_dcf_to_debt",
    "log_at",
]
continuous_feats_raw = [c for c in continuous_feats_raw if c in df.columns]

# Candidate event predictors
event_feats = [
    "evt_divcut","evt_divsusp","evt_divinit",
    "evt_liq_squeeze","evt_quick_squeeze",
    "evt_ebitdadrop","evt_cfdrop",
]
event_feats = [c for c in event_feats if c in df.columns]

# Final model feature list (events in levels; continuous will be z-scored with␣
↪z_ prefix)
MODEL_FEATS = [f"z_{c}" for c in continuous_feats_raw] + event_feats

# Leakage audit: ensure no distress-definition variables are included (raw or␣
↪scaled variants)
leakage_hits = []
for v in DISTRESS_DEFINITION_VARS:
    if v in continuous_feats_raw or v in event_feats or f"z_{v}" in MODEL_FEATS:
        leakage_hits.append(v)

if leakage_hits:
    raise ValueError(f"Leakage audit failed: distress-definition variables␣
↪present in feature set: {leakage_hits}")

print("Continuous (to be scaled):", continuous_feats_raw)
```

```
print("Events (kept in levels):", event_feats)
print("MODEL_FEATS (post-scaling names):", MODEL_FEATS)
```

```
Continuous (to be scaled): ['sp_cfo_to_debt', 'sp_focf_to_debt',
'sp_dcf_to_debt', 'log_at']
Events (kept in levels): ['evt_divcut', 'evt_divsusp', 'evt_divinit',
'evt_liq_squeeze', 'evt_quick_squeeze', 'evt_ebitdadrop', 'evt_cfdrop']
MODEL_FEATS (post-scaling names): ['z_sp_cfo_to_debt', 'z_sp_focf_to_debt',
'z_sp_dcf_to_debt', 'z_log_at', 'evt_divcut', 'evt_divsusp', 'evt_divinit',
'evt_liq_squeeze', 'evt_quick_squeeze', 'evt_ebitdadrop', 'evt_cfdrop']
```

1.8.2 6.2 Modeling sample and target availability

[238]:
```
# Modeling requires a defined next-year label
model_mask = df["target_next_year_distress"].notna()
df_model = df.loc[model_mask].copy()

print("Modeling sample size:", df_model.shape[0])
display(df_model["split"].value_counts().to_frame("n_obs"))
```

```
Modeling sample size: 63602
```

	n_obs
split	
train	44783
test	12404
val	6415

1.8.3 6.3 Replace infinities and set up train-only median imputation for remaining NaNs

[239]:
```
# Replace inf with NaN for preprocessing
for c in continuous_feats_raw:
    df_model[c] = pd.to_numeric(df_model[c], errors="coerce").replace([np.inf,␣
↪-np.inf], np.nan)

# Train-only medians for remaining NaNs (after earlier imputation steps)
train_medians = df_model.loc[df_model["split"]=="train", continuous_feats_raw].
↪median()

for c in continuous_feats_raw:
    df_model[c] = df_model[c].fillna(train_medians[c])

# Event features: coerce to Int8 with missing -> 0 (conservative) but preserve␣
↪missingness flags separately if desired
for c in event_feats:
    df_model[c] = pd.to_numeric(df_model[c], errors="coerce").fillna(0).
↪astype("Int8")
```

```
assert df_model[continuous_feats_raw].isna().sum().sum() == 0, "NaNs remain in␣
↪continuous features after train-median fill."
```

1.8.4 6.4 Winsorize continuous features (train quantile bounds)

[240]:
```
winsor_bounds = {}
for c in continuous_feats_raw:
    lo, hi = winsorize_train_bounds(df_model.loc[df_model["split"]=="train",␣
↪c], CONFIG["WINSOR_LO_Q"], CONFIG["WINSOR_HI_Q"])
    winsor_bounds[c] = (lo, hi)
    df_model[c] = apply_bounds(df_model[c], lo, hi)

winsor_tbl = pd.DataFrame(
    [{"feature": c, "lo": winsor_bounds[c][0], "hi": winsor_bounds[c][1]} for c␣
↪in continuous_feats_raw]
)
display(winsor_tbl)
```

	feature	lo	hi
0	sp_cfo_to_debt	-921.231234	935.052976
1	sp_focf_to_debt	-1363.298366	606.936691
2	sp_dcf_to_debt	-760.216000	1106.832305
3	log_at	0.000000	17.742346

1.8.5 6.5 Standardize continuous features (train-fit scaler; z__ prefix)

[241]:
```
from sklearn.preprocessing import StandardScaler

# Standardize continuous features (fit on TRAIN only)
scaler = StandardScaler()
df_model[continuous_feats_raw] = df_model[continuous_feats_raw].apply(lambda s:␣
↪pd.to_numeric(s, errors="coerce"))

train_cont = df_model.loc[df_model["split"] == "train", continuous_feats_raw].
↪astype(float)
scaler.fit(train_cont)

Z_all = scaler.transform(df_model[continuous_feats_raw].astype(float))
for j, c in enumerate(continuous_feats_raw):
    df_model[f"z_{c}"] = Z_all[:, j].astype(float)

# Final modeling matrix (events forced to clean 0/1 ints)
z_cols = [f"z_{c}" for c in continuous_feats_raw]
X = df_model[z_cols + event_feats].copy()

for c in event_feats:
```

```
    X[c] = pd.to_numeric(X[c], errors="coerce")
    X[c] = X[c].fillna(0).astype("int8")
    assert set(X[c].unique()).issubset({0, 1}), f"{c} not binary after coercion:
 ↪ {sorted(X[c].unique())}"

y = df_model["target_next_year_distress"].astype(int)

# Split views
splits = {}
for sp in ["train", "val", "test"]:
    mask = df_model["split"] == sp
    splits[sp] = {"X": X.loc[mask, :], "y": y.loc[mask], "df": df_model.
 ↪loc[mask, :]}

print({sp: (v["X"].shape[0], v["X"].shape[1]) for sp, v in splits.items()})

# Numeric-safe finiteness check
assert np.isfinite(X.astype("float64").to_numpy()).all(), "Non-finite values in␣
 ↪modeling matrix."
```

```
{'train': (44783, 11), 'val': (6415, 11), 'test': (12404, 11)}
```

1.9 7. Model Selection & Training

1.9.1 7A. Logit model (primary baseline: calibrated PD with interpretable coefficients)

7A.1 Hyperparameter tuning on out-of-time validation year

[242]:
```
train_X, train_y = splits["train"]["X"], splits["train"]["y"]
val_X, val_y = splits["val"]["X"], splits["val"]["y"]

results = []
for C in CONFIG["LOGIT_C_GRID"]:
    mdl = LogisticRegression(C=C, solver="lbfgs", max_iter=2000,␣
 ↪random_state=SEED)
    mdl.fit(train_X, train_y)
    val_proba = mdl.predict_proba(val_X)[:, 1]
    results.append({
        "C": C,
        "val_roc_auc": roc_auc_score(val_y, val_proba),
        "val_pr_auc": average_precision_score(val_y, val_proba),
        "val_brier": brier_score_loss(val_y, val_proba),
    })

tune_tbl = pd.DataFrame(results).sort_values("val_roc_auc", ascending=False)
display(tune_tbl)

best_C = float(tune_tbl.iloc[0]["C"])
```

```
print("Selected C:", best_C)
```

```
       C  val_roc_auc  val_pr_auc  val_brier
3  10.00     0.676865    0.455770   0.151421
2   1.00     0.676852    0.455752   0.151420
1   0.10     0.676730    0.455663   0.151433
0   0.01     0.676215    0.454998   0.151521
```

```
Selected C: 10.0
```

7A.2 Fit final Logit on train+val and generate PDs for all splits

[243]:
```
trainval_mask = df_model["split"].isin(["train","val"])
X_trainval = X.loc[trainval_mask, :]
y_trainval = y.loc[trainval_mask]

logit_clf = LogisticRegression(C=best_C, solver="lbfgs", max_iter=3000,␣
 ↪random_state=SEED)
logit_clf.fit(X_trainval, y_trainval)

df_model["pd_logit"] = logit_clf.predict_proba(X)[:, 1]
df_model["pd_logit_val"] = np.where(df_model["split"]=="val",␣
 ↪df_model["pd_logit"], np.nan)
df_model["pd_logit_test"] = np.where(df_model["split"]=="test",␣
 ↪df_model["pd_logit"], np.nan)

print("Example PDs:")
display(df_model[["firm_id","fyear","label_year","split","target_next_year_distress","pd_logit
 ↪head(10))
```

```
Example PDs:
```

	firm_id	fyear	label_year	split	target_next_year_distress	pd_logit
0	10000	2014	2015	train	0	0.116327
1	10000	2015	2016	train	0	0.314120
2	10000	2016	2017	train	0	0.149576
3	10000	2017	2018	train	0	0.149310
4	10000	2018	2019	train	0	0.148268
5	10000	2019	2020	train	0	0.276771
6	10000	2020	2021	train	0	0.219111
7	10000	2021	2022	val	0	0.143384
8	10000	2022	2023	test	0	0.149862
9	10000	2023	2024	test	0	0.142376

7A.3 Inference audit (statsmodels Logit; clustered standard errors)

[244]:
```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from scipy import stats
```

```python
from statsmodels.stats.sandwich_covariance import cov_cluster,␣
↪cov_cluster_2groups
# Statsmodels requires numpy arrays; keep column names for tables.
X_sm = sm.add_constant(X_trainval, has_constant="add")
y_sm = y_trainval.astype(float)

logit_sm = sm.Logit(y_sm, X_sm)
res_sm = logit_sm.fit(disp=False, maxiter=200)

# --- Firm cluster (numeric codes to avoid dtype issues) ---
firm_raw = df_model.loc[trainval_mask, "firm_id"]
firm_codes = pd.factorize(firm_raw, sort=True)[0].astype(np.int64)

cov_firm = cov_cluster(res_sm, firm_codes)
se_firm = np.sqrt(np.diag(cov_firm))

# --- Two-way cluster (firm + year), with feasibility + shape guards ---
year_raw = df_model.loc[trainval_mask, "label_year"]
firm_raw = df_model.loc[trainval_mask, "firm_id"]

firm_codes = pd.factorize(firm_raw, sort=True)[0].astype(np.int64)
year_codes = pd.factorize(year_raw, sort=True)[0].astype(np.int64)

if (np.unique(firm_codes).size < 2) or (np.unique(year_codes).size < 2):
    # Not enough clusters in one dimension -> two-way clustering not identified
    se_2 = se_firm.copy()
else:
    cov_2 = cov_cluster_2groups(res_sm, firm_codes, year_codes)
    cov_2 = np.asarray(cov_2)

    k = len(res_sm.params)
    if cov_2.ndim == 2 and cov_2.shape == (k, k):
        se_2 = np.sqrt(np.diag(cov_2))
    elif cov_2.ndim == 1 and cov_2.size == k:
        # Some statsmodels versions may return only the diagonal variances
        se_2 = np.sqrt(cov_2)
    else:
        # Unexpected shape -> fall back (safer than crashing)
        se_2 = se_firm.copy()

coef = res_sm.params
p_firm = 2 * (1 - stats.norm.cdf(np.abs(coef / se_firm)))
p_2 = 2 * (1 - stats.norm.cdf(np.abs(coef / se_2)))

infer_tbl = pd.DataFrame({
    "coef_logodds": coef,
    "se_firm": se_firm,
```

```
    "p_firm": p_firm,
    "se_firm_year": se_2,
    "p_firm_year": p_2,
    "odds_ratio": np.exp(coef),
})
infer_tbl.index.name = "feature"
display(infer_tbl)
```

```
                    coef_logodds   se_firm        p_firm  se_firm_year  \
feature
const                  -1.780544  0.025707  0.000000e+00      0.025707
z_sp_cfo_to_debt       -0.444343  0.083829  1.154285e-07      0.083829
z_sp_focf_to_debt       0.263842  0.054067  1.061105e-06      0.054067
z_sp_dcf_to_debt        0.065225  0.042233  1.224935e-01      0.042233
z_log_at               -0.414242  0.017539  0.000000e+00      0.017539
evt_divcut             -0.064761  0.077261  4.019164e-01      0.077261
evt_divsusp             0.262571  0.105561  1.286898e-02      0.105561
evt_divinit            -0.303205  0.050937  2.639179e-09      0.050937
evt_liq_squeeze         0.945996  0.054297  0.000000e+00      0.054297
evt_quick_squeeze       0.300247  0.052808  1.303268e-08      0.052808
evt_ebitdadrop         -0.089131  0.043568  4.077501e-02      0.043568
evt_cfdrop              0.056553  0.036792  1.242652e-01      0.036792

                     p_firm_year  odds_ratio
feature
const               0.000000e+00    0.168546
z_sp_cfo_to_debt    1.154285e-07    0.641245
z_sp_focf_to_debt   1.061105e-06    1.301923
z_sp_dcf_to_debt    1.224935e-01    1.067399
z_log_at            0.000000e+00    0.660841
evt_divcut          4.019164e-01    0.937292
evt_divsusp         1.286898e-02    1.300268
evt_divinit         2.639179e-09    0.738448
evt_liq_squeeze     0.000000e+00    2.575378
evt_quick_squeeze   1.303268e-08    1.350193
evt_ebitdadrop      4.077501e-02    0.914725
evt_cfdrop          1.242652e-01    1.058182
```

7A.4 Economic magnitude: MEM marginal effects and IQR-scaled effects

[245]:
```
# MEM marginal effects using statsmodels (on train+val)
try:
    me = res_sm.get_margeff(at="mean")
    me_tbl = me.summary_frame()
    # Align to inference table index (margeff typically excludes const)
    me_tbl = me_tbl.reindex(infer_tbl.index)
    display(me_tbl)
except Exception as e:
```

```
    print("Marginal effects computation failed:", e)
    me_tbl = None

# IQR-scaled effects for continuous features (using TRAIN distribution, mapped␣
 ↪into z-space)
train_df = df_model.loc[df_model["split"]=="train", :].copy()

iqr_rows = []
for j, raw_c in enumerate(continuous_feats_raw):
    q25 = float(train_df[raw_c].quantile(0.25))
    q75 = float(train_df[raw_c].quantile(0.75))
    iqr = q75 - q25
    sd = float(scaler.scale_[j]) if scaler.scale_[j] > 0 else np.nan
    delta_z = iqr / sd if sd and not np.isnan(sd) else np.nan
    beta = float(infer_tbl.loc[f"z_{raw_c}", "coef_logodds"]) if f"z_{raw_c}"␣
 ↪in infer_tbl.index else np.nan
    logodds_delta = beta * delta_z if not np.isnan(beta) and not np.
 ↪isnan(delta_z) else np.nan
    iqr_rows.append({
        "raw_feature": raw_c,
        "IQR_raw": iqr,
        "delta_z_equiv": delta_z,
        "logodds_change_IQR": logodds_delta,
        "odds_ratio_IQR": float(np.exp(logodds_delta)) if not np.
 ↪isnan(logodds_delta) else np.nan,
    })

iqr_tbl = pd.DataFrame(iqr_rows)
display(iqr_tbl)
```

```
                       dy/dx  Std. Err.          z       Pr(>|z|)  \
feature
const                    NaN        NaN        NaN            NaN
z_sp_cfo_to_debt   -0.066796   0.009755  -6.847070   7.537805e-12
z_sp_focf_to_debt   0.039662   0.005297   7.487199   7.035908e-14
z_sp_dcf_to_debt    0.009805   0.006976   1.405583   1.598479e-01
z_log_at           -0.062271   0.001731 -35.983022  1.541932e-283
evt_divcut         -0.009735   0.010210  -0.953473   3.403503e-01
evt_divsusp         0.039471   0.015879   2.485669   1.293083e-02
evt_divinit        -0.045579   0.007877  -5.786291   7.195761e-09
evt_liq_squeeze     0.142207   0.006180  23.011403  3.583973e-117
evt_quick_squeeze   0.045135   0.006118   7.377462   1.613350e-13
evt_ebitdadrop     -0.013399   0.006490  -2.064365   3.898316e-02
evt_cfdrop          0.008501   0.005583   1.522610   1.278563e-01

                    Conf. Int. Low  Cont. Int. Hi.
feature
```

```
const                       NaN          NaN
z_sp_cfo_to_debt      -0.085916    -0.047676
z_sp_focf_to_debt      0.029280     0.050045
z_sp_dcf_to_debt      -0.003867     0.023477
z_log_at              -0.065663    -0.058879
evt_divcut            -0.029747     0.010276
evt_divsusp            0.008348     0.070594
evt_divinit           -0.061018    -0.030140
evt_liq_squeeze        0.130095     0.154319
evt_quick_squeeze      0.033144     0.057126
evt_ebitdadrop        -0.026120    -0.000678
evt_cfdrop            -0.002442     0.019444

       raw_feature   IQR_raw  delta_z_equiv  logodds_change_IQR  \
0   sp_cfo_to_debt  0.338877       0.002203           -0.000979
1  sp_focf_to_debt  0.350600       0.002011            0.000530
2   sp_dcf_to_debt  0.386551       0.002393            0.000156
3           log_at  4.937452       1.372563           -0.568573

   odds_ratio_IQR
0        0.999022
1        1.000531
2        1.000156
3        0.566333
```

7A.5 Average Partial Effects (APEs) in probability units with cluster-robust uncertainty

[246]:
```python
# APEs (Average Partial Effects) for logit model, using delta-method SEs with␣
↪cluster-robust covariances
# Notes:
# - For logit, dP/dx_j = p*(1-p)*beta_j. The APE is the sample average of this␣
↪derivative.
# - We compute APEs on the TRAIN+VAL estimation sample used in statsmodels␣
↪(X_sm, res_sm).
# - SEs use the same cluster-robust covariance matrices already computed above␣
↪(cov_firm and cov_2).

import numpy as np
import pandas as pd
from scipy import stats
def _coerce_cov(V, names):
    """Return numeric (k x k) covariance aligned to names. Fallback logic␣
↪handles common statsmodels outputs."""
    k = len(names)

    if isinstance(V, pd.DataFrame):
        V = V.reindex(index=names, columns=names).to_numpy(dtype=float)
```

```
        return V

    V = np.asarray(V)
    V = np.squeeze(V)

    # Handle 3D objects (e.g., (3,k,k) or (k,k,3)): take first covariance slice
    if V.ndim == 3:
        if V.shape[1:] == (k, k):
            V = V[0]
        elif V.shape[:2] == (k, k):
            V = V[:, :, 0]
        else:
            V = V.reshape(-1, k, k)[0]

    # Handle diagonal-only variances
    if V.ndim == 1:
        if V.size != k:
            raise ValueError(f"Unexpected 1D covariance length: {V.size}␣
↪(expected {k})")
        V = np.diag(V.astype(float))

    if V.ndim != 2 or V.shape != (k, k):
        raise ValueError(f"Unexpected covariance shape: {V.shape} (expected␣
↪{(k, k)})")

    V = V.astype(float)
    V[~np.isfinite(V)] = 0.0
    return V

# ---- Align design matrix to params order ----
X_df = X_sm if isinstance(X_sm, pd.DataFrame) else pd.DataFrame(X_sm)
b_ser = res_sm.params

names = list(b_ser.index)
X_df = X_df.loc[:, names]                  # enforce same column order
X_audit_np = X_df.to_numpy(dtype=float)

b = b_ser.to_numpy(dtype=float)
k = len(names)

# Predicted probabilities on estimation sample
eta = X_audit_np @ b
p = 1.0 / (1.0 + np.exp(-eta))
w = p * (1.0 - p)
mw = float(np.mean(w))

# APE_j = beta_j * mean(w)
```

```
ape = b * mw
if "const" in names:
    ape[names.index("const")] = np.nan

# Delta-method gradient
t = (w * (1.0 - 2.0 * p))[:, None] * X_audit_np
dmw_db = np.mean(t, axis=0)

G = np.full((k, k), np.nan, dtype=float)
for j in range(k):
    if names[j] == "const":
        continue
    g = dmw_db * b[j]
    g[j] += mw
    G[j, :] = g

# Covariances (coerce 2-way; fallback to firm)
V_firm = _coerce_cov(cov_firm, names)
if "cov_2" in globals():
    try:
        V_2 = _coerce_cov(cov_2, names)
    except Exception:
        V_2 = V_firm
else:
    V_2 = V_firm

def _se_from_V(V):
    se = np.full(k, np.nan, dtype=float)
    for j in range(k):
        if not np.all(np.isfinite(G[j, :])):
            continue
        g = G[j, :].astype(float)
        v = (g @ V @ g).item()  # scalar quadratic form
        se[j] = np.sqrt(v) if v >= 0 else np.nan
    return se

se_ape_firm = _se_from_V(V_firm)
se_ape_2 = _se_from_V(V_2)

# p-values (normal approximation)
z_firm = ape / se_ape_firm
p_ape_firm = 2 * (1 - stats.norm.cdf(np.abs(z_firm)))

z_2 = ape / se_ape_2
p_ape_2 = 2 * (1 - stats.norm.cdf(np.abs(z_2)))

ape_tbl = pd.DataFrame({
```

```
    "APE": ape,
    "se_APE_firm": se_ape_firm,
    "p_APE_firm": p_ape_firm,
    "se_APE_firm_year": se_ape_2,
    "p_APE_firm_year": p_ape_2,
}, index=pd.Index(names, name="feature"))

display(ape_tbl)

infer_tbl_with_ape = infer_tbl.join(ape_tbl, how="left")
display(infer_tbl_with_ape)
```

```
                          APE  se_APE_firm    p_APE_firm  se_APE_firm_year  \
feature
const                     NaN          NaN           NaN               NaN
z_sp_cfo_to_debt    -0.065191     0.012284  1.114496e-07          0.013962
z_sp_focf_to_debt    0.038709     0.007938  1.080554e-06          0.010490
z_sp_dcf_to_debt     0.009569     0.006189  1.220800e-01          0.005972
z_log_at            -0.060774     0.002375  0.000000e+00          0.002552
evt_divcut          -0.009501     0.011324  4.014360e-01          0.010611
evt_divsusp          0.038522     0.015466  1.274591e-02          0.005847
evt_divinit         -0.044484     0.007452  2.382563e-09          0.017640
evt_liq_squeeze      0.138789     0.007846  0.000000e+00          0.007356
evt_quick_squeeze    0.044050     0.007739  1.258522e-08          0.009161
evt_ebitdadrop      -0.013077     0.006393  4.081291e-02          0.011871
evt_cfdrop           0.008297     0.005405  1.247813e-01          0.008460

                   p_APE_firm_year
feature
const                          NaN
z_sp_cfo_to_debt      3.026941e-06
z_sp_focf_to_debt     2.242170e-04
z_sp_dcf_to_debt      1.090546e-01
z_log_at              0.000000e+00
evt_divcut            3.705832e-01
evt_divsusp           4.434098e-11
evt_divinit           1.167658e-02
evt_liq_squeeze       0.000000e+00
evt_quick_squeeze     1.521654e-06
evt_ebitdadrop        2.706563e-01
evt_cfdrop            3.267253e-01
                   coef_logodds   se_firm         p_firm  se_firm_year  \
feature
const                 -1.780544  0.025707  0.000000e+00      0.025707
z_sp_cfo_to_debt      -0.444343  0.083829  1.154285e-07      0.083829
z_sp_focf_to_debt      0.263842  0.054067  1.061105e-06      0.054067
z_sp_dcf_to_debt       0.065225  0.042233  1.224935e-01      0.042233
```

```
z_log_at             -0.414242  0.017539  0.000000e+00      0.017539
evt_divcut           -0.064761  0.077261  4.019164e-01      0.077261
evt_divsusp           0.262571  0.105561  1.286898e-02      0.105561
evt_divinit          -0.303205  0.050937  2.639179e-09      0.050937
evt_liq_squeeze       0.945996  0.054297  0.000000e+00      0.054297
evt_quick_squeeze     0.300247  0.052808  1.303268e-08      0.052808
evt_ebitdadrop       -0.089131  0.043568  4.077501e-02      0.043568
evt_cfdrop            0.056553  0.036792  1.242652e-01      0.036792

                    p_firm_year  odds_ratio       APE  se_APE_firm  \
feature
const              0.000000e+00    0.168546       NaN          NaN
z_sp_cfo_to_debt   1.154285e-07    0.641245 -0.065191     0.012284
z_sp_focf_to_debt  1.061105e-06    1.301923  0.038709     0.007938
z_sp_dcf_to_debt   1.224935e-01    1.067399  0.009569     0.006189
z_log_at           0.000000e+00    0.660841 -0.060774     0.002375
evt_divcut         4.019164e-01    0.937292 -0.009501     0.011324
evt_divsusp        1.286898e-02    1.300268  0.038522     0.015466
evt_divinit        2.639179e-09    0.738448 -0.044484     0.007452
evt_liq_squeeze    0.000000e+00    2.575378  0.138789     0.007846
evt_quick_squeeze  1.303268e-08    1.350193  0.044050     0.007739
evt_ebitdadrop     4.077501e-02    0.914725 -0.013077     0.006393
evt_cfdrop         1.242652e-01    1.058182  0.008297     0.005405

                     p_APE_firm  se_APE_firm_year  p_APE_firm_year
feature
const                       NaN               NaN              NaN
z_sp_cfo_to_debt   1.114496e-07          0.013962     3.026941e-06
z_sp_focf_to_debt  1.080554e-06          0.010490     2.242170e-04
z_sp_dcf_to_debt   1.220800e-01          0.005972     1.090546e-01
z_log_at           0.000000e+00          0.002552     0.000000e+00
evt_divcut         4.014360e-01          0.010611     3.705832e-01
evt_divsusp        1.274591e-02          0.005847     4.434098e-11
evt_divinit        2.382563e-09          0.017640     1.167658e-02
evt_liq_squeeze    0.000000e+00          0.007356     0.000000e+00
evt_quick_squeeze  1.258522e-08          0.009161     1.521654e-06
evt_ebitdadrop     4.081291e-02          0.011871     2.706563e-01
evt_cfdrop         1.247813e-01          0.008460     3.267253e-01
```

7A.5 Walk-forward validation (expanding window)

[247]:
```
trainpool_df = df_model.loc[df_model["split"].isin(["train","val"]), :].copy()
years = sorted(trainpool_df["label_year"].unique().tolist())
years = [int(y) for y in years if pd.notna(y)]

N_SPLITS = 4
if len(years) < (N_SPLITS + 2):
    print("Not enough years for walk-forward validation; skipping.")
```

```
    wf_tbl = pd.DataFrame()
else:
    # Choose split points evenly across the year range (excluding last year to␣
 ↪keep a holdout)
    split_idx = np.linspace(2, len(years)-1, N_SPLITS, dtype=int)
    wf_rows = []
    for k in split_idx:
        train_years = years[:k]
        val_year = years[k]
        tr = trainpool_df["label_year"].isin(train_years)
        va = trainpool_df["label_year"].isin([val_year])

        X_tr = trainpool_df.loc[tr, [f"z_{c}" for c in continuous_feats_raw] +␣
 ↪event_feats]
        y_tr = trainpool_df.loc[tr, "target_next_year_distress"].astype(int)
        X_va = trainpool_df.loc[va, [f"z_{c}" for c in continuous_feats_raw] +␣
 ↪event_feats]
        y_va = trainpool_df.loc[va, "target_next_year_distress"].astype(int)

        mdl = LogisticRegression(C=best_C, solver="lbfgs", max_iter=2000,␣
 ↪random_state=SEED)
        mdl.fit(X_tr, y_tr)
        p_va = mdl.predict_proba(X_va)[:, 1]

        wf_rows.append({
            "train_years_min": min(train_years),
            "train_years_max": max(train_years),
            "val_year": val_year,
            "n_train": int(len(y_tr)),
            "n_val": int(len(y_va)),
            "roc_auc": roc_auc_score(y_va, p_va),
            "pr_auc": average_precision_score(y_va, p_va),
            "brier": brier_score_loss(y_va, p_va),
        })
    wf_tbl = pd.DataFrame(wf_rows)
    display(wf_tbl)
```

```
   train_years_min  train_years_max  val_year  n_train  n_val   roc_auc  \
0             2015             2016      2017    13343   6432  0.699239
1             2015             2017      2018    19775   6337  0.707103
2             2015             2019      2020    32285   6233  0.687055
3             2015             2021      2022    44783   6415  0.676865

     pr_auc     brier
0  0.446139  0.143053
1  0.463519  0.138756
2  0.444326  0.154089
```

```
3  0.455770  0.151421
```

1.9.2 7B. Tree-based model (XGBoost; nonlinear)

Tree models capture interactions and nonlinearities that logit cannot, but they require stronger regularization and calibration discipline.

Implementation details: - Early stopping on **PR-AUC** using validation split. - Conservative depth and regularization parameters. - Cost-sensitive weighting to reflect class imbalance and FN/FP asymmetry. - **Isotonic calibration** fit on validation predictions (train-only model remains unchanged).

7B.1 Train XGBoost with early stopping (validation PR-AUC)

[248]:
```
# Build DMatrix objects
X_tr = splits["train"]["X"]
y_tr = splits["train"]["y"].astype(int)
X_va = splits["val"]["X"]
y_va = splits["val"]["y"].astype(int)
X_te = splits["test"]["X"]
y_te = splits["test"]["y"].astype(int)

n_pos = int(y_tr.sum())
n_neg = int((y_tr==0).sum())
imbalance = (n_neg / max(n_pos, 1))

w_pos = CONFIG["COST_FN"] * imbalance
w_neg = CONFIG["COST_FP"]

w_tr = np.where(y_tr.values==1, w_pos, w_neg).astype(float)
w_va = np.where(y_va.values==1, w_pos, w_neg).astype(float)

dtrain = xgb.DMatrix(X_tr, label=y_tr, weight=w_tr, feature_names=X_tr.columns.
 ↪tolist())
dval   = xgb.DMatrix(X_va, label=y_va, weight=w_va, feature_names=X_tr.columns.
 ↪tolist())
dall   = xgb.DMatrix(X, label=y.astype(int), feature_names=X_tr.columns.
 ↪tolist())

evals = [(dtrain, "train"), (dval, "val")]

xgb_model = xgb.train(
    params=CONFIG["XGB_PARAMS"],
    dtrain=dtrain,
    num_boost_round=CONFIG["XGB_NUM_BOOST_ROUND"],
    evals=evals,
    early_stopping_rounds=CONFIG["XGB_EARLY_STOPPING"],
    verbose_eval=False,
)
```

```
print("Best iteration:", xgb_model.best_iteration)
```

```
Best iteration: 290
```

7B.2 Isotonic calibration on validation set (probability calibration)

[249]:
```
# Raw probabilities (uncalibrated)
p_val_raw = xgb_model.predict(dval)
p_all_raw = xgb_model.predict(dall)

# Fit isotonic on validation only
iso = IsotonicRegression(out_of_bounds="clip")
iso.fit(p_val_raw, y_va.values.astype(int))

df_model["pd_tree_raw"] = p_all_raw
df_model["pd_tree"] = iso.transform(p_all_raw)

print("Calibration fitted on validation only.")
display(df_model[["split","pd_tree_raw","pd_tree"]].groupby("split").mean())
```

```
Calibration fitted on validation only.
```

split	pd_tree_raw	pd_tree
test	0.785063	0.227178
train	0.769638	0.221828
val	0.776941	0.217147

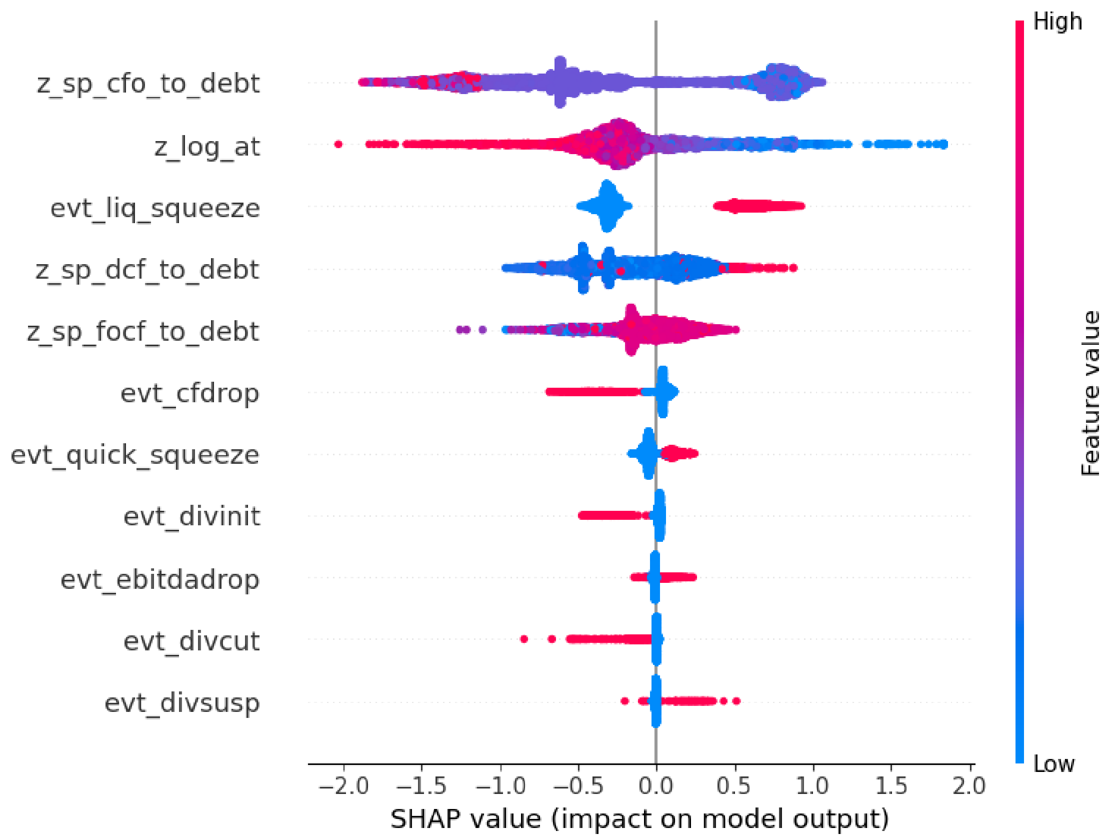
7B.3 Feature importance and SHAP (optional explainability)

[250]:
```
# Gain-based feature importance
importance = xgb_model.get_score(importance_type="gain")
imp_tbl = (pd.DataFrame({"feature": list(importance.keys()), "gain":␣
↪list(importance.values())})
             .sort_values("gain", ascending=False))
display(imp_tbl.head(20))

# Optional: SHAP summary for a subsample (can be expensive on large panels)
try:
    import shap
    shap.initjs()
    sample_n = min(5000, X_tr.shape[0])
    X_sample = X_tr.sample(sample_n, random_state=SEED)
    explainer = shap.TreeExplainer(xgb_model)
    shap_values = explainer.shap_values(X_sample)
    plt.figure()
    shap.summary_plot(shap_values, X_sample, show=False)
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / "shap_summary_tree.png", dpi=160)
```

```
    plt.show()
except Exception as e:
    print("SHAP skipped:", e)
```

```
               feature        gain
7      evt_liq_squeeze  331.876190
0      z_sp_cfo_to_debt  157.184113
8    evt_quick_squeeze  107.056656
2     z_sp_dcf_to_debt   95.077530
3             z_log_at   71.370796
1    z_sp_focf_to_debt   59.999290
10          evt_cfdrop   48.669601
6          evt_divinit   33.638363
4           evt_divcut   30.029495
5          evt_divsusp   26.450878
9       evt_ebitdadrop   22.277409
```

```
<IPython.core.display.HTML object>
```



1.10 8. Model Evaluation & Diagnostic Monitoring

All evaluation in this section treats the **test split as untouchable**: no tuning based on test results.

We report: - ROC-AUC, PR-AUC, Brier score, - calibration curve and calibration slope (reliability), - persistence benchmark, - collinearity and drift diagnostics.

1.10.1 8.1 Out-of-sample metrics (val and test) + persistence benchmark

[251]:
```
def eval_metrics(y_true: pd.Series, p: np.ndarray) -> dict:
    y_true = y_true.astype(int).values
    return {
        "roc_auc": roc_auc_score(y_true, p),
        "pr_auc": average_precision_score(y_true, p),
        "brier": brier_score_loss(y_true, p),
        "mean_p": float(np.mean(p)),
        "event_rate": float(np.mean(y_true)),
    }

rows = []
for sp in ["val","test"]:
    mask = df_model["split"] == sp
    y_sp = df_model.loc[mask, "target_next_year_distress"].astype(int)

    rows.append({"split": sp, "model":"logit", **eval_metrics(y_sp, df_model.
 ↪loc[mask, "pd_logit"].values)})
    rows.append({"split": sp, "model":"tree_calibrated", **eval_metrics(y_sp,␣
 ↪df_model.loc[mask, "pd_tree"].values)})
    rows.append({"split": sp, "model":"tree_raw", **eval_metrics(y_sp, df_model.
 ↪loc[mask, "pd_tree_raw"].values)})

    # Persistence benchmark: predict next-year distress = current-year␣
 ↪distress_dummy_t
    pers = pd.to_numeric(df_model.loc[mask, "distress_dummy_t"],␣
 ↪errors="coerce").fillna(0).astype(int).values
    rows.append({"split": sp, "model":"persistence", **eval_metrics(y_sp,␣
 ↪pers)})

metrics_tbl = pd.DataFrame(rows).sort_values(["split","model"])
display(metrics_tbl)
```

	split	model	roc_auc	pr_auc	brier	mean_p	event_rate
4	test	logit	0.684691	0.444296	0.151705	0.206673	0.217510
7	test	persistence	0.786057	0.522609	0.141325	0.209852	0.217510
5	test	tree_calibrated	0.813212	0.554200	0.129130	0.227178	0.217510
6	test	tree_raw	0.813669	0.571512	0.472277	0.785063	0.217510
0	val	logit	0.677171	0.457008	0.151299	0.201686	0.217147

```
3   val      persistence  0.780521  0.530108  0.135308  0.192518    0.217147
1   val  tree_calibrated  0.814882  0.582518  0.125667  0.217147    0.217147
2   val         tree_raw  0.811978  0.594037  0.463619  0.776941    0.217147
```

1.10.2 8.1b Early-warning vs Surveillance decomposition (state-conditional evaluation)

[252]:
```python
# Early-warning vs surveillance evaluation:
#   - Early warning: subset with distress_dummy_t == 0 (not currently␣
↪distressed)
#   - Surveillance: subset with distress_dummy_t == 1 (currently distressed)
# Also add a state-only baseline: predict next-year distress using current␣
↪distress_dummy_t only.

import numpy as np
import pandas as pd

def safe_eval_metrics(y_true: pd.Series, p: np.ndarray) -> dict:
    y = y_true.astype(int).values
    out = {
        "roc_auc": np.nan,
        "pr_auc": np.nan,
        "brier": brier_score_loss(y, p),
        "mean_p": float(np.mean(p)),
        "event_rate": float(np.mean(y)),
        "n": int(len(y)),
    }
    if np.unique(y).size >= 2:
        out["roc_auc"] = roc_auc_score(y, p)
        out["pr_auc"] = average_precision_score(y, p)
    return out

def eval_segment(df_seg: pd.DataFrame, split_name: str, segment_name: str) ->␣
↪list:
    rows = []
    if df_seg.empty:
        return rows

    y = df_seg["target_next_year_distress"].astype(int)

    # State-only baseline (uses current distress state only)
    state = pd.to_numeric(df_seg["distress_dummy_t"], errors="coerce").
↪fillna(0).astype(int).values
    base = safe_eval_metrics(y, state)

    # Models
    for col, mdl in [("pd_logit","logit"),
```

```
                      ("pd_tree","tree_calibrated"),
                      ("pd_tree_raw","tree_raw")]:
        met = safe_eval_metrics(y, df_seg[col].values)

        rows.append({
            "split": split_name,
            "segment": segment_name,
            "model": mdl,
            **met,
            "baseline_roc_auc": base["roc_auc"],
            "baseline_pr_auc": base["pr_auc"],
            "baseline_brier": base["brier"],
            "delta_roc_auc": (met["roc_auc"] - base["roc_auc"]) if (met["roc_auc"]==met["roc_auc"] and base["roc_auc"]==base["roc_auc"]) else np.nan,
            "delta_pr_auc": (met["pr_auc"] - base["pr_auc"]) if (met["pr_auc"]==met["pr_auc"] and base["pr_auc"]==base["pr_auc"]) else np.nan,
            "delta_brier": met["brier"] - base["brier"],  # negative is improvement
        })

    # Add baseline as a row for reference
    rows.append({
        "split": split_name,
        "segment": segment_name,
        "model": "state_only",
        **base,
        "baseline_roc_auc": np.nan,
        "baseline_pr_auc": np.nan,
        "baseline_brier": np.nan,
        "delta_roc_auc": 0.0,
        "delta_pr_auc": 0.0,
        "delta_brier": 0.0,
    })
    return rows

seg_rows = []
for sp in ["val", "test"]:
    df_sp = df_model.loc[df_model["split"]==sp, :].copy()

    # Only evaluate segments where current distress state is observed.
    dcur = pd.to_numeric(df_sp["distress_dummy_t"], errors="coerce")
    df_sp = df_sp.loc[dcur.notna(), :].copy()
    df_sp["distress_t_int"] = dcur.loc[dcur.notna()].astype(int)
```

```python
    seg_rows += eval_segment(df_sp.loc[df_sp["distress_t_int"]==0, :], sp, "early_warning (distress_t=0)")
    seg_rows += eval_segment(df_sp.loc[df_sp["distress_t_int"]==1, :], sp, "surveillance (distress_t=1)")

seg_metrics_tbl = pd.DataFrame(seg_rows)

if not seg_metrics_tbl.empty:
    seg_metrics_tbl = seg_metrics_tbl.sort_values(["split","segment","model"])
    display(seg_metrics_tbl)
else:
    print("No segment metrics computed (empty segments).")
```

```
   split                      segment            model   roc_auc    pr_auc  \
8   test  early_warning (distress_t=0)           logit  0.619841  0.150138
11  test  early_warning (distress_t=0)      state_only  0.500000  0.094276
9   test  early_warning (distress_t=0)  tree_calibrated  0.713255  0.193311
10  test  early_warning (distress_t=0)        tree_raw  0.713589  0.198700
12  test   surveillance (distress_t=1)           logit  0.602456  0.768443
15  test   surveillance (distress_t=1)      state_only  0.500000  0.681521
13  test   surveillance (distress_t=1)  tree_calibrated  0.634905  0.774373
14  test   surveillance (distress_t=1)        tree_raw  0.634580  0.782330
0    val  early_warning (distress_t=0)           logit  0.602689  0.139525
3    val  early_warning (distress_t=0)      state_only  0.500000  0.099035
1    val  early_warning (distress_t=0)  tree_calibrated  0.706377  0.193404
2    val  early_warning (distress_t=0)        tree_raw  0.701002  0.193748
4    val   surveillance (distress_t=1)           logit  0.624248  0.818962
7    val   surveillance (distress_t=1)      state_only  0.500000  0.712551
5    val   surveillance (distress_t=1)  tree_calibrated  0.669696  0.831021
6    val   surveillance (distress_t=1)        tree_raw  0.663411  0.834070

       brier    mean_p  event_rate     n  baseline_roc_auc  baseline_pr_auc  \
8   0.095646  0.181598    0.094276  9801               0.5         0.094276
11  0.094276  0.000000    0.094276  9801               NaN              NaN
9   0.092438  0.161276    0.094276  9801               0.5         0.094276
10  0.522983  0.743210    0.094276  9801               0.5         0.094276
12  0.362784  0.301086    0.681521  2603               0.5         0.681521
15  0.318479  1.000000    0.681521  2603               NaN              NaN
13  0.267285  0.475315    0.681521  2603               0.5         0.681521
14  0.281355  0.942650    0.681521  2603               0.5         0.681521
0   0.098722  0.177976    0.099035  5180               0.5         0.099035
3   0.099035  0.000000    0.099035  5180               NaN              NaN
1   0.093756  0.154810    0.099035  5180               0.5         0.099035
2   0.514478  0.738039    0.099035  5180               0.5         0.099035
4   0.371825  0.301135    0.712551  1235               0.5         0.712551
7   0.287449  1.000000    0.712551  1235               NaN              NaN
5   0.259510  0.478612    0.712551  1235               0.5         0.712551
```

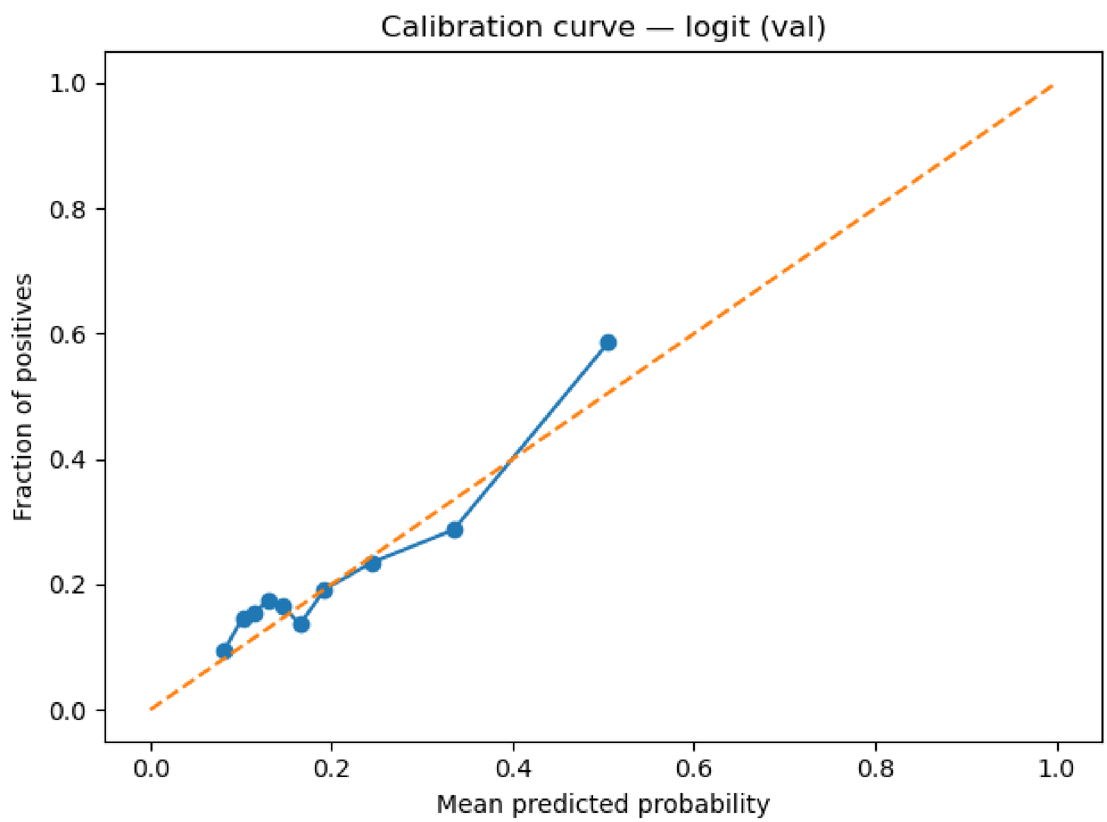
```
6   0.250302  0.940108    0.712551  1235              0.5          0.712551

    baseline_brier  delta_roc_auc  delta_pr_auc  delta_brier
8         0.094276       0.119841      0.055862     0.001370
11             NaN       0.000000      0.000000     0.000000
9         0.094276       0.213255      0.099035    -0.001838
10        0.094276       0.213589      0.104423     0.428707
12        0.318479       0.102456      0.086922     0.044305
15             NaN       0.000000      0.000000     0.000000
13        0.318479       0.134905      0.092852    -0.051193
14        0.318479       0.134580      0.100808    -0.037124
0         0.099035       0.102689      0.040490    -0.000313
3              NaN       0.000000      0.000000     0.000000
1         0.099035       0.206377      0.094370    -0.005278
2         0.099035       0.201002      0.094713     0.415443
4         0.287449       0.124248      0.106411     0.084375
7              NaN       0.000000      0.000000     0.000000
5         0.287449       0.169696      0.118470    -0.027939
6         0.287449       0.163411      0.121520    -0.037147
```

1.10.3 8.2 Calibration diagnostics (curve + calibration-in-the-large + slope)

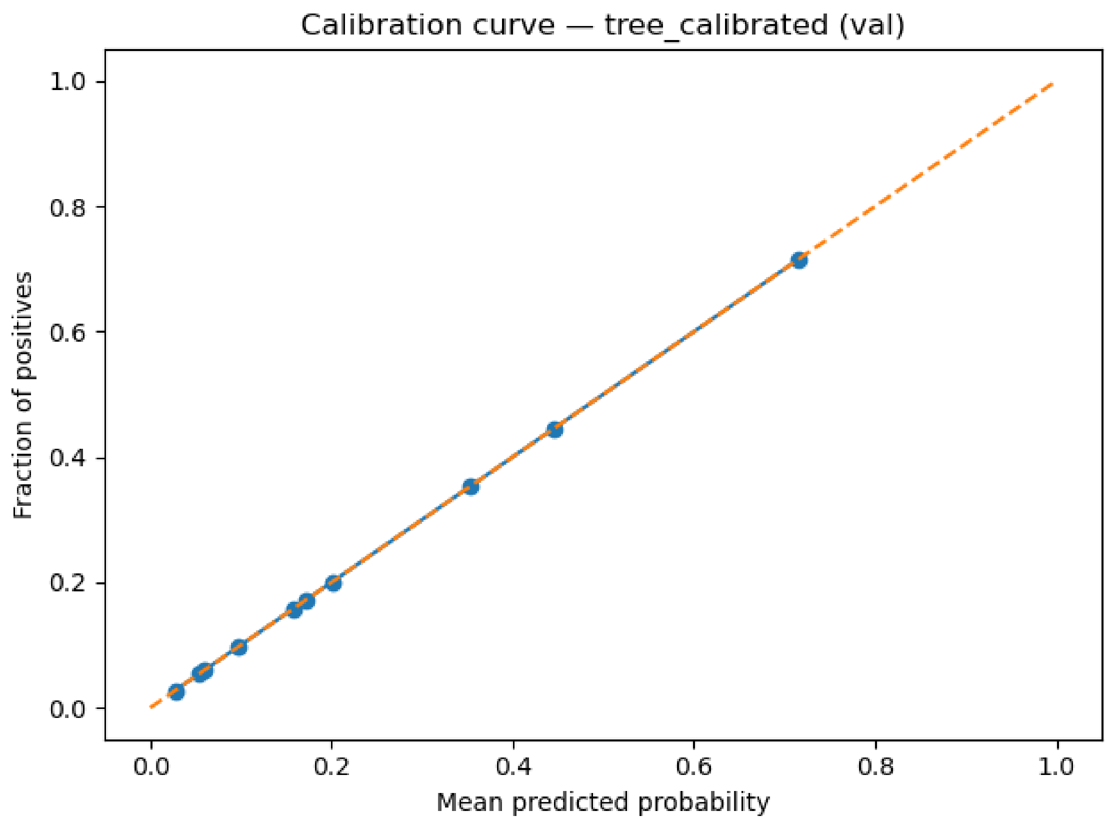
[253]:
```
def calibration_slope_intercept(y_true: np.ndarray, p: np.ndarray) ->
 ↪tuple[float,float]:
    z = logit(p)
    Xc = sm.add_constant(z, has_constant="add")
    mdl = sm.GLM(y_true, Xc, family=sm.families.Binomial())
    res = mdl.fit()
    intercept, slope = res.params[0], res.params[1]
    return float(intercept), float(slope)

def plot_calibration(y_true: np.ndarray, p: np.ndarray, title: str, fname: str):
    frac_pos, mean_pred = calibration_curve(y_true, p, n_bins=10,
 ↪strategy="quantile")
    plt.figure()
    plt.plot(mean_pred, frac_pos, marker="o")
    plt.plot([0,1],[0,1], linestyle="--")
    plt.xlabel("Mean predicted probability")
    plt.ylabel("Fraction of positives")
    plt.title(title)
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / fname, dpi=160)
    plt.show()

for sp in ["val","test"]:
    mask = df_model["split"] == sp
    y_sp = df_model.loc[mask, "target_next_year_distress"].astype(int).values
```

```
for model_name, pcol in [("logit","pd_logit"),␣
↪("tree_calibrated","pd_tree")]:
        p = df_model.loc[mask, pcol].values
        icpt, slope = calibration_slope_intercept(y_sp, p)
        print(f"{sp} | {model_name}: calibration intercept={icpt:.3f},␣
↪slope={slope:.3f}")
        plot_calibration(y_sp, p, f"Calibration curve — {model_name} ({sp})",␣
↪f"cal_curve_{model_name}_{sp}.png")
```

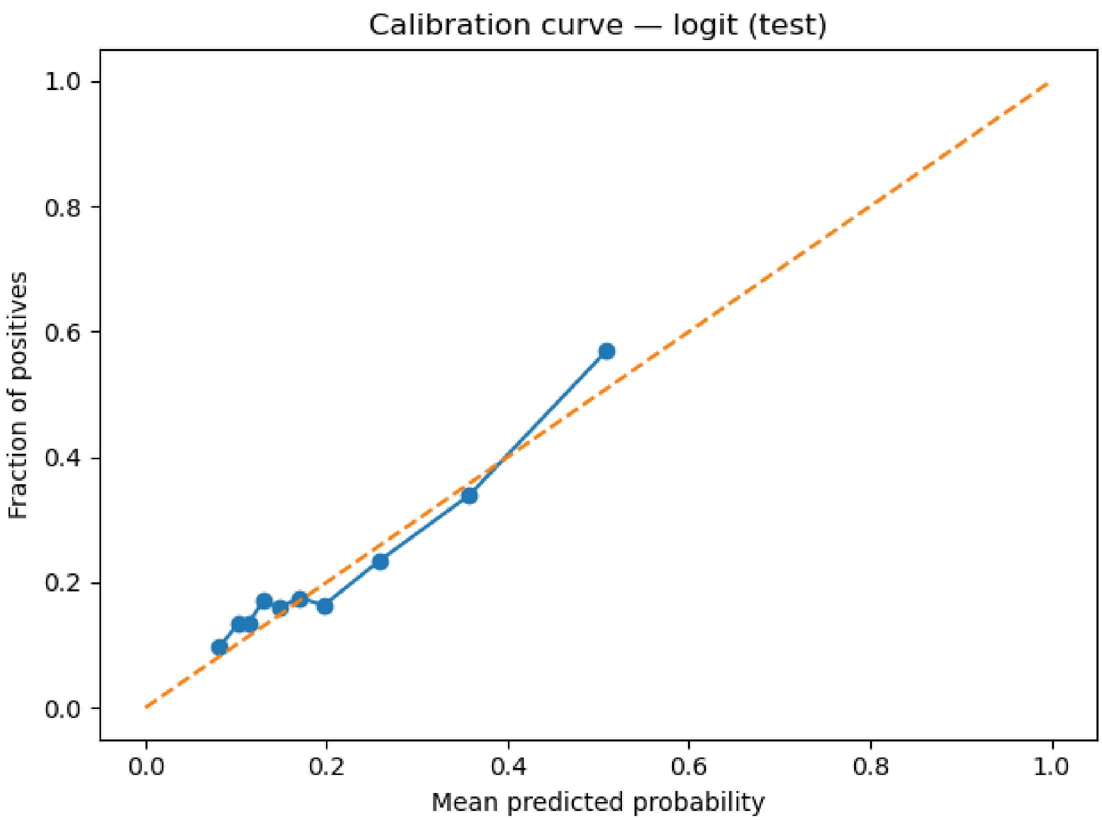
```
val | logit: calibration intercept=0.097, slope=0.995
```



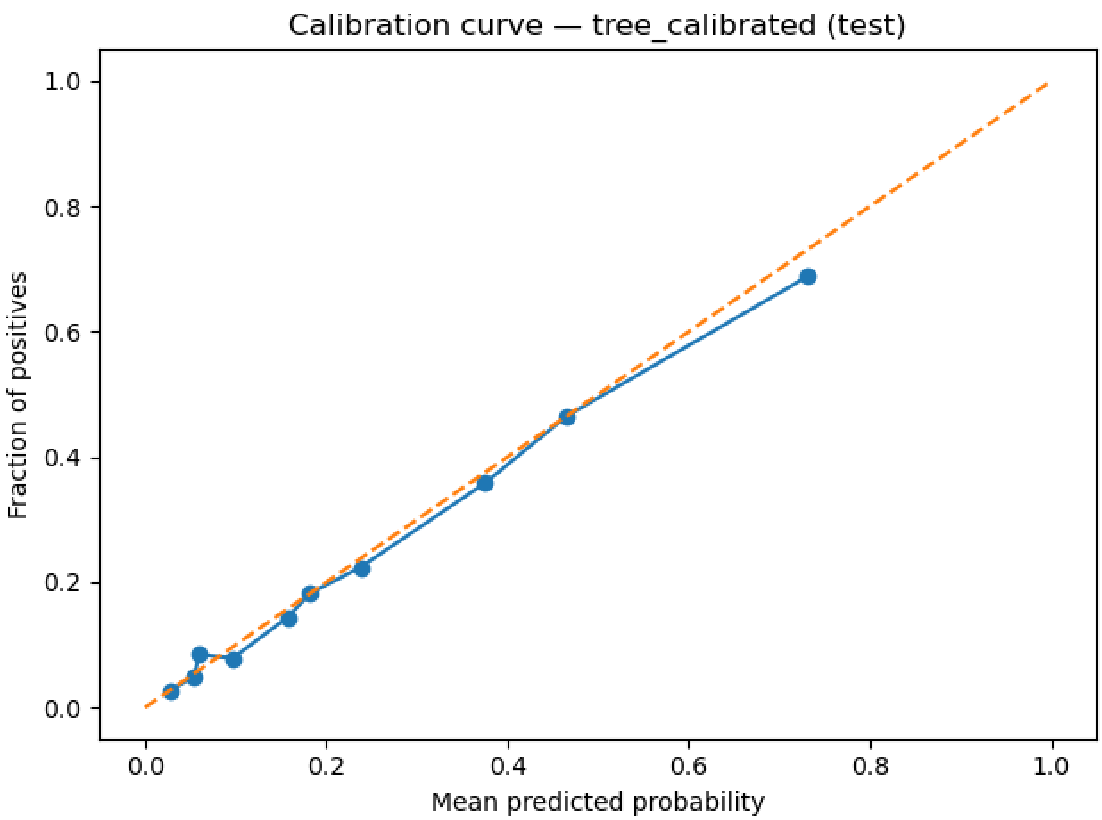

```
val | tree_calibrated: calibration intercept=0.000, slope=1.000
```




```
test | logit: calibration intercept=0.050, slope=0.981
```




```
test | tree_calibrated: calibration intercept=-0.127, slope=0.946
```

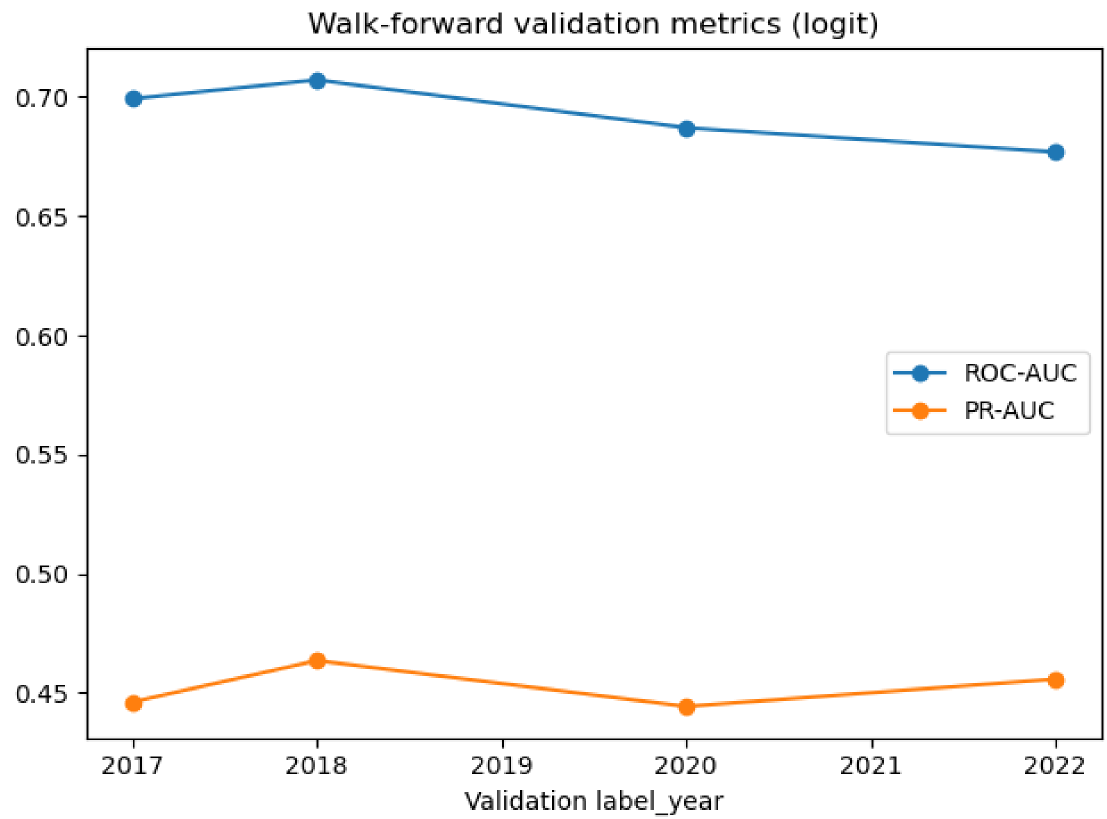


1.10.4 8.3 Temporal stability (walk-forward fold metrics)

[254]:
```
if 'wf_tbl' in globals() and len(wf_tbl) > 0:
    display(wf_tbl)
    plt.figure()
    plt.plot(wf_tbl["val_year"], wf_tbl["roc_auc"], marker="o", label="ROC-AUC")
    plt.plot(wf_tbl["val_year"], wf_tbl["pr_auc"], marker="o", label="PR-AUC")
    plt.title("Walk-forward validation metrics (logit)")
    plt.xlabel("Validation label_year")
    plt.legend()
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / "walkforward_metrics_logit.png",␣
↪dpi=160)
    plt.show()
```

```
   train_years_min  train_years_max  val_year  n_train  n_val   roc_auc  \
0             2015             2016      2017    13343   6432  0.699239
1             2015             2017      2018    19775   6337  0.707103
2             2015             2019      2020    32285   6233  0.687055
3             2015             2021      2022    44783   6415  0.676865
```

```
     pr_auc     brier
0  0.446139  0.143053
1  0.463519  0.138756
2  0.444326  0.154089
3  0.455770  0.151421
```



1.10.5 8.4 Collinearity checks (VIF + high-correlation pairs)

[255]:
```
# VIF on continuous z-features (train only)
X_vif = splits["train"]["X"][[f"z_{c}" for c in continuous_feats_raw]].copy()
X_vif = sm.add_constant(X_vif, has_constant="add")

vif_rows = []
for i, col in enumerate(X_vif.columns):
    if col == "const":
        continue
    vif_rows.append({"feature": col, "VIF":␣
↪float(variance_inflation_factor(X_vif.values, i))})

vif_tbl = pd.DataFrame(vif_rows).sort_values("VIF", ascending=False)
display(vif_tbl)
```

```python
# Correlation screen (continuous only)
corr = splits["train"]["X"][[f"z_{c}" for c in continuous_feats_raw]].corr()
high_pairs = []
for i in range(len(corr.columns)):
    for j in range(i+1, len(corr.columns)):
        v = corr.iloc[i,j]
        if abs(v) >= 0.85:
            high_pairs.append((corr.columns[i], corr.columns[j], float(v)))
high_pairs_tbl = pd.DataFrame(high_pairs, columns=["feat1","feat2","corr"]).
↪sort_values("corr", key=np.abs, ascending=False)
display(high_pairs_tbl)
```

	feature	VIF
0	z_sp_cfo_to_debt	17.726818
2	z_sp_dcf_to_debt	12.872471
1	z_sp_focf_to_debt	2.838250
3	z_log_at	1.015086

	feat1	feat2	corr
0	z_sp_cfo_to_debt	z_sp_dcf_to_debt	0.953714

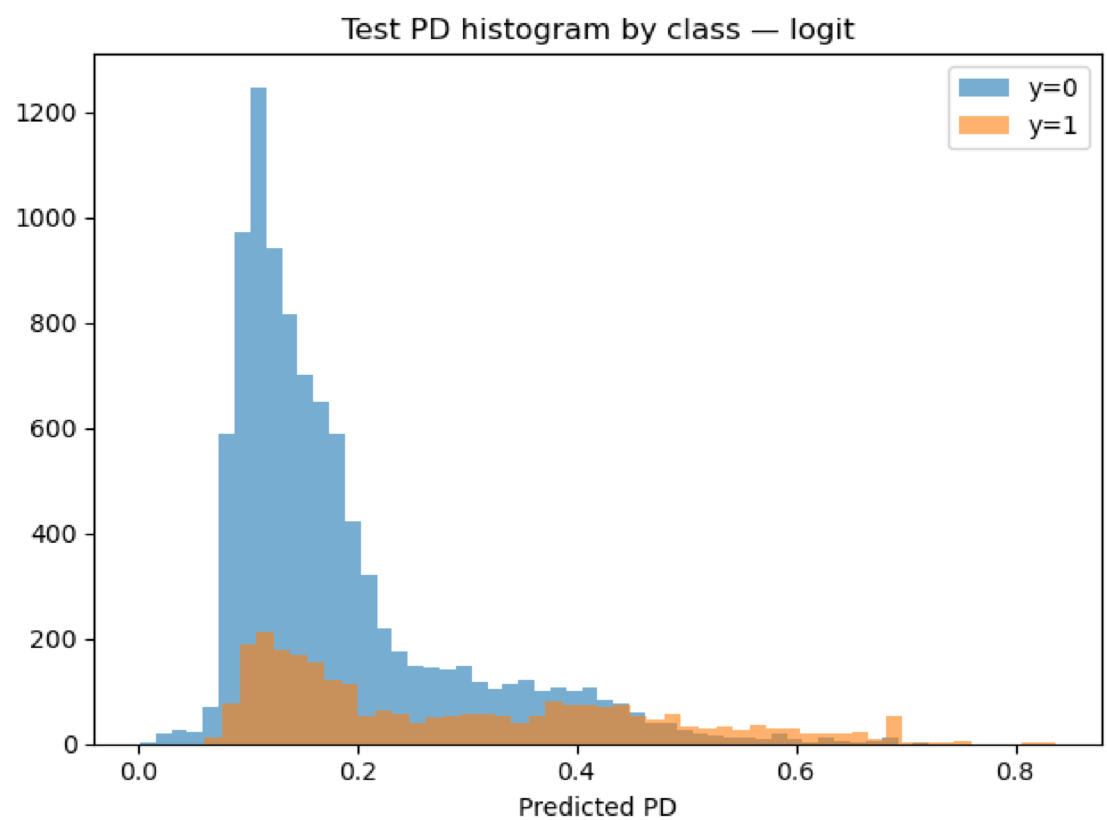
1.10.6 8.5 Drift diagnostics (standardized mean difference: train vs test)

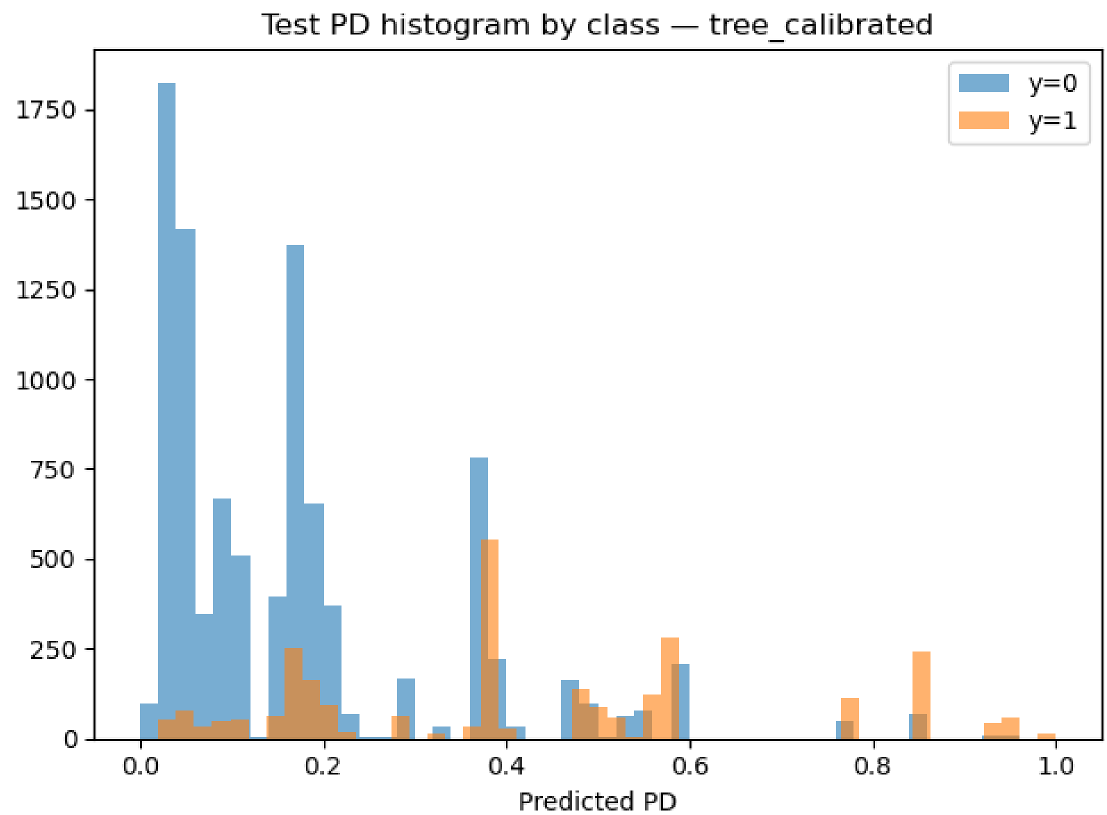
[256]:
```python
feat_cols = [f"z_{c}" for c in continuous_feats_raw] + event_feats
drift_rows = []
for c in feat_cols:
    smd = compute_smd(df_model.loc[df_model["split"]=="train", c], df_model.
↪loc[df_model["split"]=="test", c])
    drift_rows.append({"feature": c, "SMD_train_vs_test": smd})
drift_tbl = pd.DataFrame(drift_rows).sort_values("SMD_train_vs_test",␣
↪key=lambda s: s.abs(), ascending=False)
display(drift_tbl.head(25))
```

	feature	SMD_train_vs_test
6	evt_divinit	0.203455
2	z_sp_dcf_to_debt	0.113484
0	z_sp_cfo_to_debt	0.091657
10	evt_cfdrop	-0.087803
3	z_log_at	-0.084108
9	evt_ebitdadrop	-0.047233
4	evt_divcut	0.028819
1	z_sp_focf_to_debt	0.027354
7	evt_liq_squeeze	0.025131
8	evt_quick_squeeze	0.022559
5	evt_divsusp	0.006291

1.10.7 8.6 Probability distributions by class (test split)

[257]:
```
mask = df_model["split"]=="test"
y_true = df_model.loc[mask, "target_next_year_distress"].astype(int)

for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    p = df_model.loc[mask, pcol]
    plt.figure()
    plt.hist(p[y_true==0], bins=50, alpha=0.6, label="y=0")
    plt.hist(p[y_true==1], bins=50, alpha=0.6, label="y=1")
    plt.title(f"Test PD histogram by class — {model_name}")
    plt.xlabel("Predicted PD")
    plt.legend()
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / f"pd_hist_{model_name}_test.png",␣
↪dpi=160)
    plt.show()
```





1.11 9. Decision Support Layer

This section operationalizes predicted probabilities into interpretable risk signals and policies: - **Event lift tables:** how discrete events shift conditional distress risk. - **Risk deciles:** realized risk by predicted PD decile (reliability in operational tiers). - **Cost curves & capacity rules:** choose thresholds under explicit misclassification costs and screening capacity. - **Pro-forma scenarios:** sensitivity of PD to accounting-consistent adjustments (illustrative; not causal).

1.11.1 9.1 Event lift tables (prevalence, conditional risk, lift)

[258]:
```
def event_lift_table(df_in: pd.DataFrame, events: list[str], y_col: str) -> pd.
↪DataFrame:
    base = df_in[y_col].astype(float).mean()
    rows=[]
    for e in events:
        if e not in df_in.columns:
            continue
        s = pd.to_numeric(df_in[e], errors="coerce").fillna(0).astype(int)
        if s.sum() == 0:
            continue
        rate = df_in.loc[s==1, y_col].astype(float).mean()
```

```python
        prev = s.mean()
        rows.append({
            "event": e,
            "prevalence": prev,
            "cond_distress_rate": rate,
            "lift_vs_base": rate/base if base>0 else np.nan,
            "base_rate": base,
            "n_event": int(s.sum()),
        })
    out = pd.DataFrame(rows).sort_values("lift_vs_base", ascending=False)
    return out

for sp in ["train","test"]:
    df_sp = df_model.loc[df_model["split"]==sp, :]
    print(f"\nEvent lift - {sp}")
    display(event_lift_table(df_sp, event_feats, "target_next_year_distress").
 ↪head(20))
```

```
Event lift - train

               event  prevalence  cond_distress_rate  lift_vs_base  base_rate  \
3     evt_liq_squeeze    0.251144            0.408731      1.950992   0.209499
4   evt_quick_squeeze    0.279414            0.375689      1.793274   0.209499
1         evt_divsusp    0.016100            0.217753      1.039399   0.209499
6          evt_cfdrop    0.119465            0.192897      0.920754   0.209499
5      evt_ebitdadrop    0.093049            0.182145      0.869433   0.209499
0          evt_divcut    0.052520            0.172194      0.821931   0.209499
2         evt_divinit    0.076502            0.130181      0.621391   0.209499

   n_event
3    11247
4    12513
1      721
6     5350
5     4167
0     2352
2     3426


Event lift - test

               event  prevalence  cond_distress_rate  lift_vs_base  base_rate  \
3     evt_liq_squeeze    0.240326            0.415297      1.909319    0.21751
4   evt_quick_squeeze    0.269349            0.371146      1.706338    0.21751
1         evt_divsusp    0.015318            0.200000      0.919496    0.21751
5      evt_ebitdadrop    0.107223            0.181203      0.833077    0.21751
6          evt_cfdrop    0.149387            0.180788      0.831169    0.21751
2         evt_divinit    0.030877            0.154047      0.708228    0.21751
```

```
0         evt_divcut    0.046275             0.149826      0.688821    0.21751

   n_event
3     2981
4     3341
1      190
5     1330
6     1853
2      383
0      574
```

1.11.2 9.2 Event transitions (0→1 activation; 1→1 persistence)

[259]:
```python
def transition_stats(df_in: pd.DataFrame, event: str) -> dict:
    s = pd.to_numeric(df_in[event], errors="coerce").fillna(0).astype(int)
    s_l1 = df_in.groupby("firm_id")[event].shift(1)
    s_l1 = pd.to_numeric(s_l1, errors="coerce").fillna(0).astype(int)

    act_01 = ((s_l1==0) & (s==1)).mean()
    pers_11 = ((s_l1==1) & (s==1)).mean()
    return {"event": event, "activation_01_rate": float(act_01),␣
↪"persistence_11_rate": float(pers_11)}

rows=[]
for e in event_feats:
    rows.append(transition_stats(df_model, e))
trans_tbl = pd.DataFrame(rows).sort_values("activation_01_rate",␣
↪ascending=False)
display(trans_tbl)
```

	event	activation_01_rate	persistence_11_rate
4	evt_quick_squeeze	0.143407	0.130986
3	evt_liq_squeeze	0.128345	0.117166
6	evt_cfdrop	0.114682	0.014402
5	evt_ebitdadrop	0.086538	0.009135
2	evt_divinit	0.063583	0.000000
0	evt_divcut	0.045517	0.006226
1	evt_divsusp	0.017043	0.000000

1.11.3 9.3 Risk deciles (expected vs realized distress by PD tier)

[260]:
```python
def decile_table(df_in: pd.DataFrame, p_col: str, y_col: str) -> pd.DataFrame:
    d = df_in[[p_col, y_col]].dropna().copy()
    d["decile"] = pd.qcut(d[p_col], 10, labels=False, duplicates="drop") + 1
    out = d.groupby("decile").agg(
        n=("decile","size"),
        mean_pd=(p_col,"mean"),
        realized_rate=(y_col,"mean"),
```

```
    ).reset_index()
    out["calibration_gap"] = out["realized_rate"] - out["mean_pd"]
    return out

for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    print(f"\nTest deciles - {model_name}")
    dt = decile_table(df_model.loc[df_model["split"]=="test", :], pcol,␣
 ↪"target_next_year_distress")
    display(dt)
```

```
Test deciles - logit
```

	decile	n	mean_pd	realized_rate	calibration_gap
0	1	1241	0.081628	0.096696	0.015069
1	2	1240	0.102016	0.133065	0.031048
2	3	1240	0.114409	0.135484	0.021074
3	4	1241	0.129569	0.170024	0.040455
4	5	1240	0.148127	0.159677	0.01155
5	6	1240	0.169695	0.175	0.005305
6	7	1241	0.197234	0.163578	-0.033657
7	8	1240	0.257899	0.233871	-0.024028
8	9	1240	0.357376	0.33871	-0.018666
9	10	1241	0.508704	0.568896	0.060192

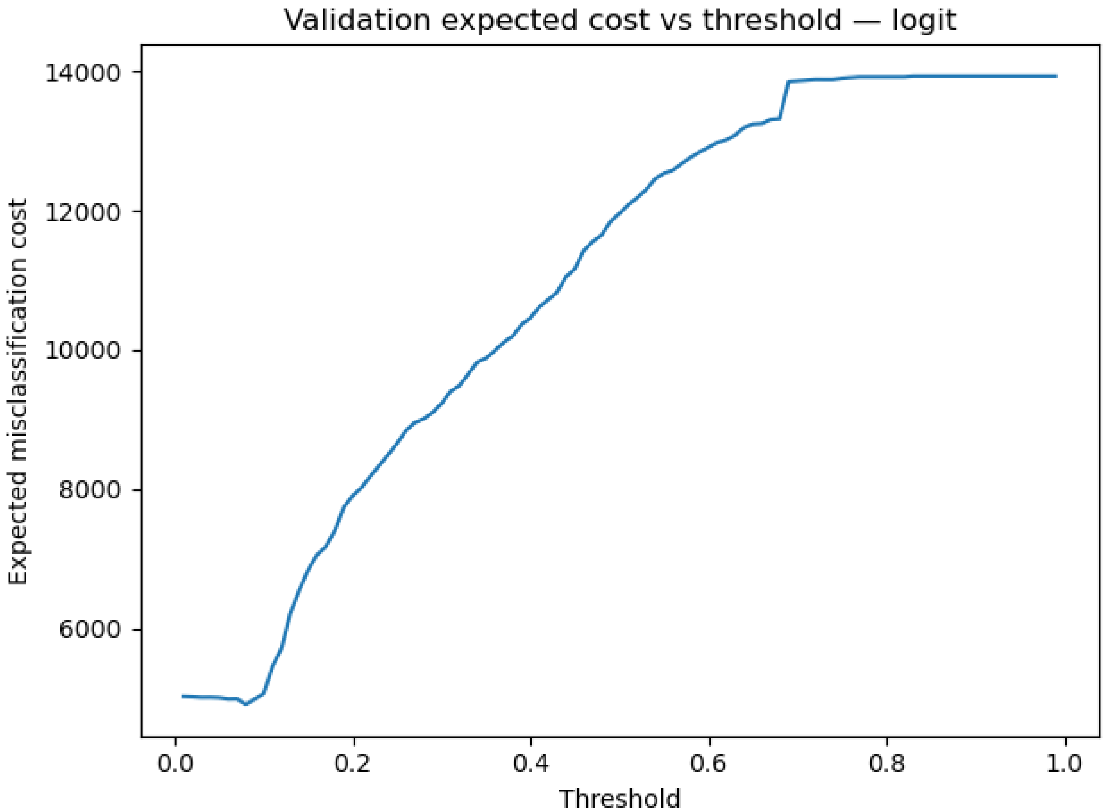
```
Test deciles - tree_calibrated
```

	decile	n	mean_pd	realized_rate	calibration_gap
0	1	1858	0.028195	0.027449	-0.000746
1	2	1612	0.053999	0.049628	-0.004371
2	3	378	0.060230	0.084656	0.024426
3	4	1274	0.096365	0.078493	-0.017872
4	5	1316	0.157347	0.143617	-0.01373
5	6	1575	0.182338	0.182222	-0.000116
6	7	791	0.237828	0.223767	-0.014061
7	8	1267	0.375230	0.358327	-0.016903
8	9	1243	0.465166	0.465004	-0.000162
9	10	1090	0.731150	0.688073	-0.043077

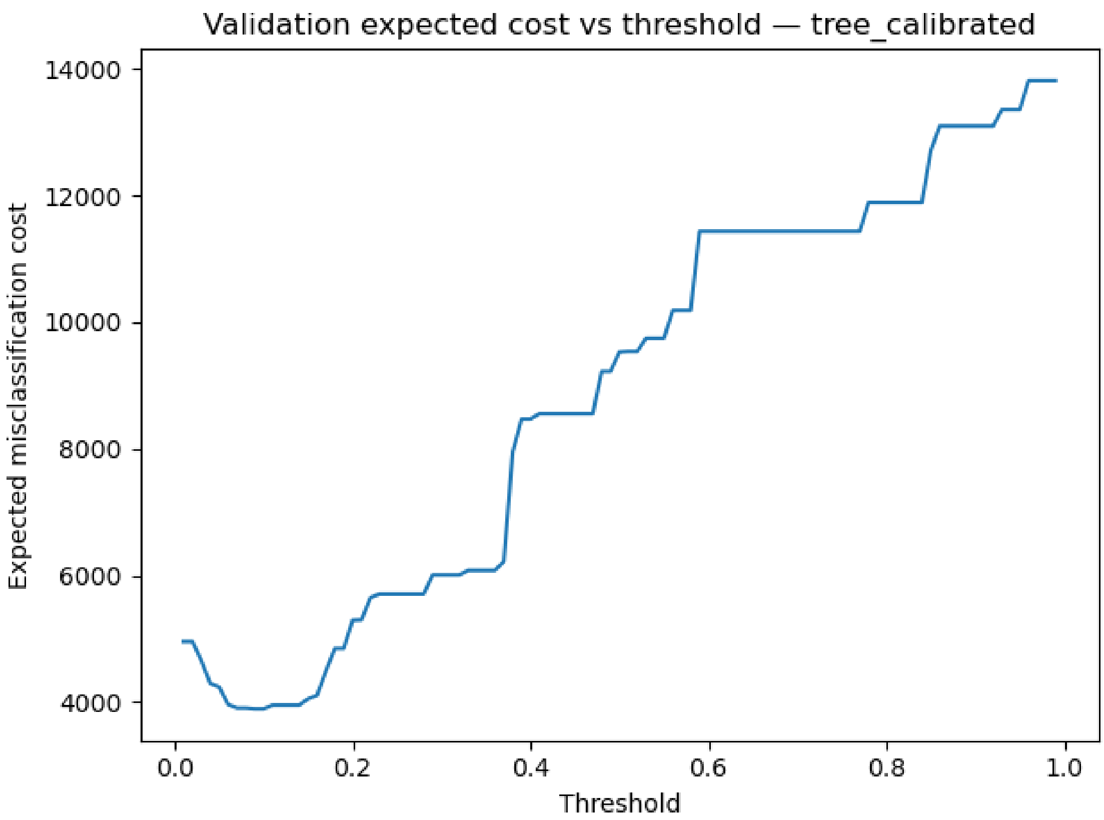
1.11.4 9.4 Cost curves and threshold selection (validation-only)

[261]:
```
COST_FN = float(CONFIG["COST_FN"])
COST_FP = float(CONFIG["COST_FP"])
CAPACITY_PCT = float(CONFIG["CAPACITY_PCT"])

def expected_cost(y_true: np.ndarray, p: np.ndarray, thr: float) -> float:
    y_hat = (p >= thr).astype(int)
    tn, fp, fn, tp = confusion_matrix(y_true, y_hat).ravel()
```

```
    return COST_FN*fn + COST_FP*fp

def cost_curve(y_true: np.ndarray, p: np.ndarray, grid: np.ndarray) -> pd.
 ↪DataFrame:
    rows=[]
    for thr in grid:
        y_hat = (p >= thr).astype(int)
        tn, fp, fn, tp = confusion_matrix(y_true, y_hat).ravel()
        rows.append({
            "thr": float(thr),
            "TP": int(tp), "FP": int(fp), "TN": int(tn), "FN": int(fn),
            "cost": float(COST_FN*fn + COST_FP*fp),
            "tpr": float(tp/(tp+fn)) if (tp+fn)>0 else np.nan,
            "fpr": float(fp/(fp+tn)) if (fp+tn)>0 else np.nan,
        })
    return pd.DataFrame(rows)

grid = np.linspace(0.01, 0.99, 99)

mask = df_model["split"]=="val"
y_val_np = df_model.loc[mask, "target_next_year_distress"].astype(int).values

thr_tbls = {}
for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    p = df_model.loc[mask, pcol].values
    cc = cost_curve(y_val_np, p, grid)
    thr_star = float(cc.loc[cc["cost"].idxmin(), "thr"])
    thr_capacity = float(np.quantile(p, 1-CAPACITY_PCT))
    thr_tbls[model_name] = {"thr_cost_opt": thr_star, "thr_capacity":␣
 ↪thr_capacity}

    plt.figure()
    plt.plot(cc["thr"], cc["cost"])
    plt.title(f"Validation expected cost vs threshold - {model_name}")
    plt.xlabel("Threshold")
    plt.ylabel("Expected misclassification cost")
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / f"cost_curve_{model_name}_val.png",␣
 ↪dpi=160)
    plt.show()

    print(model_name, "cost-opt thr=", thr_star, "capacity thr=", thr_capacity)

display(pd.DataFrame(thr_tbls).T)
```




```
logit cost-opt thr= 0.08 capacity thr= 0.285228215001878
```



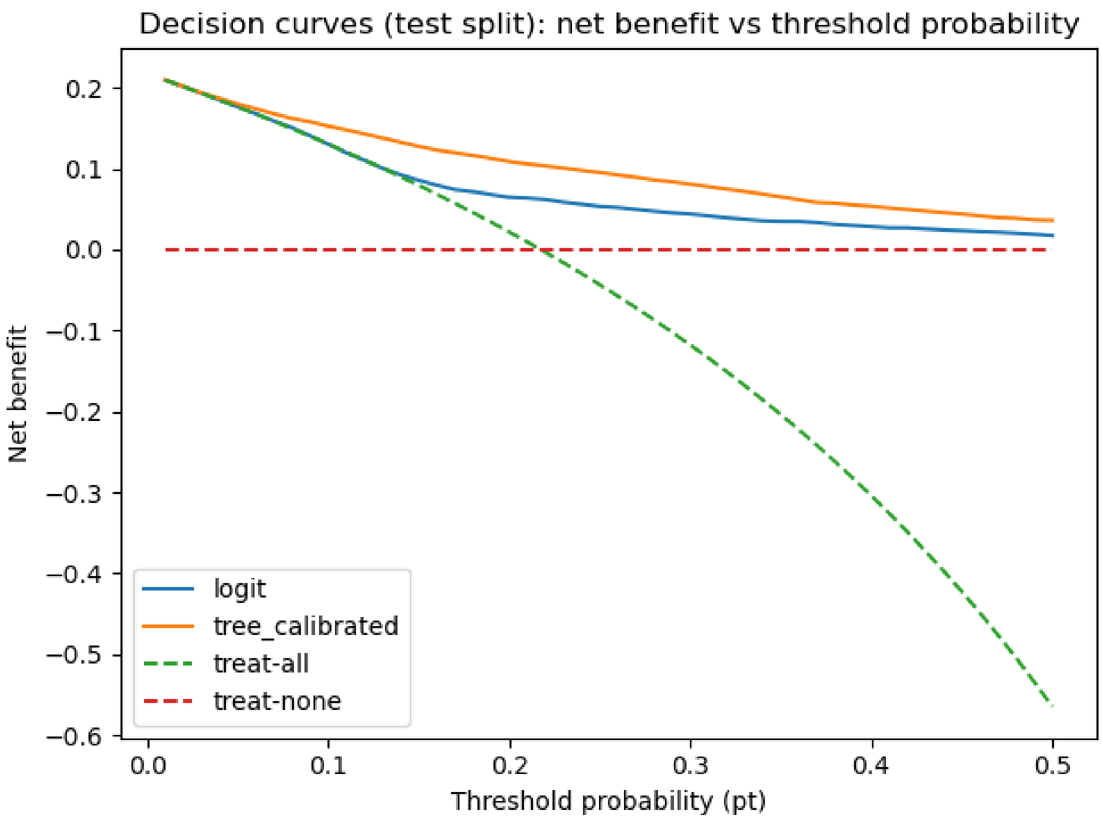

```
tree_calibrated cost-opt thr= 0.09 capacity thr= 0.37842777371406555
```

	thr_cost_opt	thr_capacity
logit	0.08	0.285228
tree_calibrated	0.09	0.378428

1.11.5 9.5 Decision curve analysis (net benefit)

[262]:
```
def net_benefit(y_true: np.ndarray, p: np.ndarray, pt: float) -> float:
    y_hat = (p >= pt).astype(int)
    tn, fp, fn, tp = confusion_matrix(y_true, y_hat).ravel()
    n = len(y_true)
    w = pt/(1-pt)
    return (tp/n) - (fp/n)*w

mask = df_model["split"]=="test"
y_test_np = df_model.loc[mask, "target_next_year_distress"].astype(int).values

pts = np.linspace(0.01, 0.50, 50)
plt.figure()
for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
```

```
    p = df_model.loc[mask, pcol].values
    nb = [net_benefit(y_test_np, p, pt) for pt in pts]
    plt.plot(pts, nb, label=model_name)

# Treat-all and treat-none baselines
event_rate = y_test_np.mean()
nb_all = [event_rate - (1-event_rate)*(pt/(1-pt)) for pt in pts]
nb_none = [0 for _ in pts]
plt.plot(pts, nb_all, linestyle="--", label="treat-all")
plt.plot(pts, nb_none, linestyle="--", label="treat-none")

plt.title("Decision curves (test split): net benefit vs threshold probability")
plt.xlabel("Threshold probability (pt)")
plt.ylabel("Net benefit")
plt.legend()
plt.tight_layout()
plt.savefig(Path(CONFIG["FIG_DIR"]) / "decision_curves_test.png", dpi=160)
plt.show()
```



1.11.6 9.6 Scenario analysis (accounting-consistent pro-forma adjustments; illustrative)

[263]:
```
# Scenario engine: recompute a single-row feature vector using the same rules␣
↪as the main pipeline.

def build_model_features_from_row(row: pd.Series) -> pd.DataFrame:
    # --- recompute continuous features ---
    at = row.get("at", np.nan)
    dlc = row.get("dlc", np.nan)
    dltt = row.get("dltt", np.nan)
    seq = row.get("seq", np.nan)
    mibt = row.get("mibt", 0.0)
    oibdp = row.get("oibdp", np.nan)
    oancf = row.get("oancf", np.nan)
    xint = row.get("xint", np.nan)
    capx = row.get("capx", np.nan)

    total_debt = np.nansum([dlc, dltt])
    equity_plus_mi = np.nansum([seq, mibt])
    total_cap = np.nansum([total_debt, equity_plus_mi])

    # log transforms
    log_at = np.log(at) if pd.notna(at) and at > 0 else np.nan

    # ratios
    sp_cfo_to_debt = (oancf/total_debt) if pd.notna(oancf) and pd.
↪notna(total_debt) and total_debt != 0 else np.nan
    ffo_proxy = (oancf + xint) if pd.notna(oancf) and pd.notna(xint) else np.nan
    if pd.notna(ffo_proxy) and "txp" in row.index and pd.notna(row.get("txp",␣
↪np.nan)):
        ffo_proxy = ffo_proxy - row.get("txp", np.nan)

    # keep computed but excluded from MODEL_FEATS
    sp_ffo_to_debt = (ffo_proxy/total_debt) if pd.notna(ffo_proxy) and pd.
↪notna(total_debt) and total_debt != 0 else np.nan

    if pd.notna(capx):
        focf_proxy = oancf - capx if pd.notna(oancf) else np.nan
        sp_focf_to_debt = (focf_proxy/total_debt) if pd.notna(focf_proxy) and␣
↪total_debt != 0 else np.nan
    else:
        sp_focf_to_debt = np.nan

    dcf_proxy = (oancf + xint) if pd.notna(oancf) and pd.notna(xint) else np.nan
    sp_dcf_to_debt = (dcf_proxy/total_debt) if pd.notna(dcf_proxy) and␣
↪total_debt != 0 else np.nan
```

```python
    cont = {
        "sp_cfo_to_debt": sp_cfo_to_debt,
        "sp_focf_to_debt": sp_focf_to_debt,
        "sp_dcf_to_debt": sp_dcf_to_debt,
        "log_at": log_at,
    }

    # --- events (must match earlier definitions) ---
    # Dividend events
    evt_divcut = 0
    evt_divsusp = 0
    evt_divinit = 0
    if div_col is not None:
        dv = abs(row.get(div_col, np.nan)) if pd.notna(row.get(div_col, np.nan)) else np.nan
        dv_l1 = row.get("dv_obs_l1", np.nan)  # precomputed lag in df
        if pd.notna(dv_l1) and dv_l1 > 1e-2 and pd.notna(dv):
            div_pct_change = (dv - dv_l1)/dv_l1
            evt_divcut = int(div_pct_change <= event_params.get("DIV_CUT_THR_P10_BOUNDED", -0.25))
        if pd.notna(dv_l1) and dv_l1 > 0 and (pd.isna(dv) or dv <= 1e-4):
            evt_divsusp = 1
        if (pd.isna(dv_l1) or dv_l1 <= 1e-4) and pd.notna(dv) and dv > 1e-4:
            evt_divinit = 1

    # Liquidity events
    act = row.get("act", np.nan)
    lct = row.get("lct", np.nan)
    invt = row.get("invt", np.nan)

    current_ratio = (act/lct) if pd.notna(act) and pd.notna(lct) and lct != 0 else np.nan
    quick_ratio = ((act - invt)/lct) if pd.notna(act) and pd.notna(invt) and pd.notna(lct) and lct != 0 else current_ratio

    evt_liq_squeeze = int(pd.notna(current_ratio) and current_ratio < 1.0)
    evt_quick_squeeze = int(pd.notna(quick_ratio) and quick_ratio < 0.8)

    # EBITDA/CFO drop using lag values stored on the row (from df)
    ebitda = row.get("oibdp", np.nan)
    ebitda_l1 = row.get("ebitda_l1", np.nan)
    evt_ebitdadrop = int(pd.notna(ebitda_l1) and ebitda_l1 > 0 and ((pd.notna(ebitda) and ebitda <= 0) or (pd.notna(ebitda) and (ebitda/ebitda_l1) < 0.5)))

    cfo = row.get("oancf", np.nan)
```

```python
    cfo_l1 = row.get("cfo_l1", np.nan)
    evt_cfdrop = int(pd.notna(cfo_l1) and cfo_l1 > 0 and ((pd.notna(cfo) and␣
↪cfo <= 0) or (pd.notna(cfo) and (cfo/cfo_l1) < 0.5)))

    events = {
        "evt_divcut": evt_divcut,
        "evt_divsusp": evt_divsusp,
        "evt_divinit": evt_divinit,
        "evt_liq_squeeze": evt_liq_squeeze,
        "evt_quick_squeeze": evt_quick_squeeze,
        "evt_ebitdadrop": evt_ebitdadrop,
        "evt_cfdrop": evt_cfdrop,
    }

    # Assemble into 1-row DF with raw continuous + events
    out = {}
    for c in continuous_feats_raw:
        out[c] = cont.get(c, np.nan)
    for e in event_feats:
        out[e] = events.get(e, 0)
    out = pd.DataFrame([out])

    # Preprocessing: train medians, winsor, scaler -> z_
    for c in continuous_feats_raw:
        v = out[c].replace([np.inf, -np.inf], np.nan)
        v = v.fillna(train_medians[c])
        lo, hi = winsor_bounds[c]
        v = apply_bounds(v, lo, hi)
        out[c] = v

    Z = scaler.transform(out[continuous_feats_raw].astype(float))
    for j, c in enumerate(continuous_feats_raw):
        out[f"z_{c}"] = Z[:, j]

    # Final feature vector in MODEL_FEATS order
    return out[[f"z_{c}" for c in continuous_feats_raw] + event_feats]

def predict_pd_from_features(X_row: pd.DataFrame) -> dict:
    pd_logit = float(logit_clf.predict_proba(X_row)[:, 1][0])
    drow = xgb.DMatrix(X_row, feature_names=X_row.columns.tolist())
    pd_tree_raw = float(xgb_model.predict(drow)[0])
    pd_tree = float(iso.transform([pd_tree_raw])[0])
    return {"pd_logit": pd_logit, "pd_tree": pd_tree, "pd_tree_raw":␣
↪pd_tree_raw}

# Select a representative high-risk test observation
test_df = df_model.loc[df_model["split"]=="test", :].copy()
```

```
rep_idx = test_df["pd_logit"].idxmax()
row0 = df.loc[rep_idx, :]  # use df (feature-engineered, imputed), not df_model
base_X = build_model_features_from_row(row0)
base_pd = predict_pd_from_features(base_X)

print("Representative observation (highest logit PD in test):")
display(df_model.loc[rep_idx,␣
↪["firm_id","fyear","label_year","pd_logit","pd_tree","target_next_year_distress"]])
print("Base PDs:", base_pd)

# Scenario 1: Liquidity buffer to current ratio = 1.2 (increase current assets;␣
↪illustrative)
row1 = row0.copy()
if "act" in row1.index and "lct" in row1.index and pd.notna(row1["act"]) and pd.
↪notna(row1["lct"]) and row1["lct"] > 0:
    target_cr = 1.2
    add_act = max(0.0, target_cr*row1["lct"] - row1["act"])
    row1["act"] = row1["act"] + add_act
    if "che" in row1.index and pd.notna(row1.get("che", np.nan)):
        row1["che"] = row1["che"] + add_act  # assume added liquidity goes to␣
↪cash
X1 = build_model_features_from_row(row1)
pd1 = predict_pd_from_features(X1)

# Scenario 2: CFO improvement of +10% of assets (accounting-consistent in the␣
↪short-run is debatable; treat as stress-test)
row2 = row0.copy()
if "oancf" in row2.index and "at" in row2.index and pd.notna(row2["at"]):
    delta = 0.10 * row2["at"]
    row2["oancf"] = (row2["oancf"] if pd.notna(row2.get("oancf", np.nan)) else␣
↪0.0) + delta
X2 = build_model_features_from_row(row2)
pd2 = predict_pd_from_features(X2)

scenario_tbl = pd.DataFrame([
    {"scenario":"base", **base_pd},
    {"scenario":"liquidity_buffer_CR_1.2", **pd1},
    {"scenario":"CFO_plus_10pct_assets", **pd2},
])
display(scenario_tbl)
```

```
Representative observation (highest logit PD in test):

firm_id                         29816
fyear                            2022
label_year                       2023
pd_logit                      0.83702
pd_tree                      0.949633
```

```
target_next_year_distress          1
Name: 49786, dtype: object

Base PDs: {'pd_logit': 0.8370197058301744, 'pd_tree': 0.9496333599090576,
'pd_tree_raw': 0.9942809343338013}

                  scenario  pd_logit   pd_tree  pd_tree_raw
0                     base  0.837020  0.949633     0.994281
1  liquidity_buffer_CR_1.2  0.596257  0.477707     0.966974
2    CFO_plus_10pct_assets  0.838468  0.928571     0.994172
```

1.12 10. Results Summary & Interpretation Guardrails

1.12.1 10.1 Interpretation guardrails (publication-ready language)

- The label is a **constructed proxy** for balance-sheet/coverage stress; it is not a legal default outcome.
- Coefficients and SHAP values are **associational and predictive**, not causal effects.
- Even with leakage controls, residual mechanical endogeneity may remain because accounting choices jointly affect both predictors and the proxy label.
- Attrition (missing next-year observations) can create sample-selection distortions; diagnostics are reported via `has_next_year_obs`.

1.12.2 10.2 Replication artifacts

The following tables/exports are written to `outputs/` for downstream paper workflow: - `config_summary.json` - `distress_rule.json` - `event_dictionary.csv` - `logit_inference_table.csv` - `metrics_table.csv` - `predictions.csv`

1.12.3 10.3 Export tables, thresholds, and predictions

[264]:
```python
out_dir = Path(CONFIG["OUTPUT_DIR"])

# Config + distress rule
(out_dir / "config_summary.json").write_text(json.dumps(CONFIG, indent=2))
(out_dir / "distress_rule.json").write_text(json.dumps(DISTRESS_RULE, indent=2))

# Event dictionary
event_dict.to_csv(out_dir / "event_dictionary.csv", index=False)

# Logit inference table
infer_tbl.reset_index().to_csv(out_dir / "logit_inference_table.csv",␣
↪index=False)

# Metrics table
metrics_tbl.to_csv(out_dir / "metrics_table.csv", index=False)

# Predictions export (replication-friendly)
export_cols =␣
↪["firm_id","gvkey","fyear","label_year","split","target_next_year_distress","pd_logit","pd_
```

```
export_cols = [c for c in export_cols if c in df_model.columns]
export_cols += [c for c in event_feats if c in df_model.columns]
pred_export = df_model[export_cols].copy()
pred_export.to_csv(out_dir / "predictions.csv", index=False)

print("Wrote artifacts to:", out_dir.resolve())
print_df(pred_export, n=10, name="predictions.csv preview")
```

```
Wrote artifacts to: /Users/test/Desktop/Test Models/AIinFinance/outputs

predictions.csv preview (top 10 rows):

  firm_id  gvkey  fyear  label_year  split  target_next_year_distress  \
0   10000  10000   2014        2015  train                          0
1   10000  10000   2015        2016  train                          0
2   10000  10000   2016        2017  train                          0
3   10000  10000   2017        2018  train                          0
4   10000  10000   2018        2019  train                          0
5   10000  10000   2019        2020  train                          0
6   10000  10000   2020        2021  train                          0
7   10000  10000   2021        2022    val                          0
8   10000  10000   2022        2023   test                          0
9   10000  10000   2023        2024   test                          0

   pd_logit   pd_tree  evt_divcut  evt_divsusp  evt_divinit  evt_liq_squeeze  \
0  0.116327  0.020202           0            0            1                0
1  0.314120  0.060241           0            0            0                1
2  0.149576  0.035088           0            0            0                0
3  0.149310  0.035088           0            0            0                0
4  0.148268  0.050000           0            0            0                0
5  0.276771  0.087571           0            0            0                0
6  0.219111  0.060241           0            0            0                0
7  0.143384  0.058296           0            0            0                0
8  0.149862  0.214844           0            0            0                0
9  0.142376  0.107595           0            0            0                0

   evt_quick_squeeze  evt_ebitdadrop  evt_cfdrop
0                  1               0           0
1                  1               0           0
2                  1               0           0
3                  1               0           0
4                  1               0           0
5                  1               0           0
6                  0               0           0
7                  1               0           0
8                  1               0           1
9                  1               0           0
```

1.12.4 10.4 Deployment and maintenance (future work)

This notebook produces a research-grade replication pipeline. For production use (not required for journal replication), a minimal MLOps extension would include: - scheduled re-scoring and monitoring for drift in feature distributions and target prevalence, - retraining triggers and versioned model registry, - data validation contracts (schema + unit tests) for the upstream Compustat extraction process.