Part A — Load & Prepare (Price-aware)

What: Load Sprint-2 data, map columns, ensure we have a usable price (or an estimate), and basic comparability fields. Why: Cart optimisation is price-driven; we need price/PPU ready and apples-to-apples fields (subcategory, unit, size band). Logic: Use raw price when present; if missing, estimate from PPU × size × pack.

```
# A0) Imports & settings
import re, numpy as np, pandas as pd
from pathlib import Path
pd.set_option("display.max_colwidth", 120)

# >>> Update the file path if needed
DATA = Path(r"C:/Shazza/T2 2025/Cap Stone
B/Sprint_2_Model_dev/smart_substitution_dataset 8.csv")

# A1) Load (CSV/XLSX)
if DATA.suffix.lower() in {".xlsx", ".xls"}:
    df = pd.read_excel(DATA)
else:
    df = pd.read_csv(DATA)

# normalise cols
df.columns = [c.strip().lower().replace(" ","_") for c in df.columns]

def pick(cols):
    for c in cols:
        if c in df.columns: return c
    return None

COL = {
    "code":  pick(["product_code","code","barcode","sku_id","id"]),
    "name":  pick(["name","item_name","title","product_name"]),
    "brand": pick(["brand_clean","brand","brand_name"]),

"pbrand":pick(["brand_tier","parent_brand","brand_parent","brand_group
"]),
    "cat":   pick(["category","cat"]),
    "sub":
pick(["subcategory","sub_cat","category_level_2","cat_lvl2"]),
    "unit":  pick(["unit_type","uom","unit"]),
    "qty":
pick(["std_item_size","unit_qty","quantity","qty","size_value"]),
    "pack":  pick(["pack_count","pack","pack_qty"]),
    "price": pick(["price","current_price","sale_price","price_now"]),
    "ppu":   pick(["price_per_unit","ppu","unit_price"]),
    "band":  pick(["size_band","band"]),
}
```

```
# A2) Safe defaults
if COL["pack"] is None: df["pack_count"] = 1; COL["pack"] =
"pack_count"
if COL["qty"]  is None: df["std_item_size"] = np.nan; COL["qty"]  =
"std_item_size"

# A3) Ensure size band
def size_band_from_qty(q):
    if pd.isna(q): return np.nan
    q = float(q)
    if q < 250: return "Small"
    if q < 750: return "Medium"
    return "Large"

if COL["band"] is None:
    df["size_band"] = df[COL["qty"]].map(size_band_from_qty)
    COL["band"] = "size_band"

# A4) Price fields: prefer raw price; else estimate pack price from
PPU × size × pack
if COL["price"] is None and COL["ppu"] is not None:
    df["price_est_single"] = df[COL["ppu"]] * df[COL["qty"]]
    df["price_est_pack"]   = df["price_est_single"] * df[COL["pack"]]
    PRICE_ITEM = "price_est_pack"
else:
    PRICE_ITEM = COL["price"]  # real shelf price
PRICE_UNIT = COL["ppu"]

# A5) Quick glance
df[[COL["name"], COL["sub"], COL["unit"], COL["qty"], COL["pack"],
PRICE_ITEM, PRICE_UNIT]].head(3)
```

```
                                          name subcategory
unit_type  \
0       coles hot cross buns traditional fruit      easter      each

1                   coles hot cross buns choc chip      easter      each

2  coles hot cross buns traditional fruit mini      easter      each


   std_item_size  pack_count  sale_price  price_per_unit
0            6.0           1         3.0            0.73
1            6.0           1         3.0            0.73
2            9.0           1         3.0            0.49
```

Part B — Price-First Rules Baseline

What: Only consider comparable items, then pick the cheapest. Why: Guarantees sensible swaps and maximises savings without training. Logic: Same subcategory & unit; size band ±1 (or size proximity ≥80%); sort by price (or PPU if price missing).

```
# B0) Comparability helpers
BAND_ORDER = {"Small":0,"Medium":1,"Large":2,"Mixed":1}
def band_dist(a,b): return abs(BAND_ORDER.get(str(a),1) -
BAND_ORDER.get(str(b),1))

def comparable_mask(pool, qrow, size_prox_min=0.80):
    m = pool.index != qrow.name
    m &= (pool[COL["sub"]] == qrow[COL["sub"]])
    if COL["unit"]:
        m &= (pool[COL["unit"]] == qrow.get(COL["unit"]))
    # band ±1 OR size proximity when sizes known
    band_ok = True
    if COL["band"]:
        band_ok = (pool[COL["band"]].map(lambda b: band_dist(b,
qrow.get(COL["band"]))) <= 1)
    size_ok = True
    if pd.notna(qrow.get(COL["qty"])):
        # proximity = 1 - |Δ|/query_size  (clip to [0,1])
        size_ok = ((1 -
(pool[COL["qty"]].sub(qrow.get(COL["qty"])).abs() /
max(qrow.get(COL["qty"]),1e-9)).clip(0,1)) >= size_prox_min) |
pool[COL["qty"]].isna()
    return m & (band_ok | size_ok)

# B1) Cheapest substitute
def cheapest_substitute(qidx, prefer="item"):
    """
    prefer='item': compare total price (raw price if available, else
estimated pack price)
    prefer='unit': compare PPU
    """
    q = df.loc[qidx]
    cand = df[comparable_mask(df, q)]
    if cand.empty: return None

    key = PRICE_UNIT if (prefer=="unit" and PRICE_UNIT is not None)
else PRICE_ITEM
    cand = cand[cand[key].notna()]
    if cand.empty: return None

    cheaper = cand[cand[key] < q.get(key)]
    chosen = (cheaper if not cheaper.empty else cand).sort_values(key,
ascending=True).iloc[0]
```

```
    # deltas (+ means candidate is cheaper)
    qp_item, cp_item = q.get(PRICE_ITEM), chosen.get(PRICE_ITEM)
    qp_unit, cp_unit = q.get(PRICE_UNIT), chosen.get(PRICE_UNIT)
    d_price  = (qp_item - cp_item) if pd.notna(qp_item) and
pd.notna(cp_item) else np.nan
    d_pricep = (d_price/qp_item)   if pd.notna(d_price) else np.nan
    d_ppu    = (qp_unit - cp_unit) if pd.notna(qp_unit) and
pd.notna(cp_unit) else np.nan
    d_ppup   = (d_ppu/qp_unit)     if pd.notna(d_ppu) else np.nan

    tags=[]
    if COL["band"] and chosen.get(COL["band"]) == q.get(COL["band"]):
tags.append("Same size band")
    if COL["unit"] and chosen.get(COL["unit"]) == q.get(COL["unit"]):
tags.append("Same unit")
    if pd.notna(d_pricep): tags.append(f"Cheaper by {d_pricep*100:.0f}
%")
    elif pd.notna(d_ppup): tags.append(f"Cheaper by {d_ppup*100:.0f}%
(PPU)")

    return {
        "query_idx": qidx,
        "query_name": q.get(COL["name"]),
        "query_subcat": q.get(COL["sub"]),
        "query_unit": q.get(COL["unit"]),
        "query_size": q.get(COL["qty"]),
        "query_price_item": qp_item, "query_price_unit": qp_unit,
        "cand_idx": chosen.name,
        "cand_name": chosen.get(COL["name"]),
        "cand_unit": chosen.get(COL["unit"]),
        "cand_size": chosen.get(COL["qty"]),
        "cand_price_item": cp_item, "cand_price_unit": cp_unit,
        "ΔPrice": d_price, "ΔPrice%": d_pricep,
        "ΔPPU": d_ppu,   "ΔPPU%": d_ppup,
        "is_cheaper_item": int(pd.notna(d_price) and d_price>0),
        "is_cheaper_unit": int(pd.notna(d_ppu)   and d_ppu>0),
        "tags": "; ".join(tags)
    }
```

B2 — Demo & Batch KPIs

What: Show one example and aggregate savings. Why: Quick proof we're optimising for price while staying comparable.

```
def find_items(q, limit=10):
    m = df[COL["name"]].str.contains(str(q), case=False, na=False)
    return df.loc[m, [COL["name"], COL["brand"], COL["sub"],
COL["unit"], COL["qty"], PRICE_ITEM, PRICE_UNIT]].head(limit)
```

```
# Demo: change the search term as you like
("sugar","milk","cola","tuna"…)
find_items("sugar", limit=5)
example_idx = find_items("sugar", limit=1).index[0]
cheapest_substitute(example_idx, prefer="item")
```

```
{'query_idx': 61,
 'query_name': 'noshu 95 sugar free banana bread slices',
 'query_subcat': 'bakery',
 'query_unit': 'g',
 'query_size': 240.0,
 'query_price_item': 5.9,
 'query_price_unit': 0.0246,
 'cand_idx': 59,
 'cand_name': 'tip top english muffins pizza flavoured',
 'cand_unit': 'g',
 'cand_size': 6.0,
 'cand_price_item': 3.1,
 'cand_price_unit': 0.0086,
 'ΔPrice': 2.8000000000000003,
 'ΔPrice%': 0.4745762711864407,
 'ΔPPU': 0.016,
 'ΔPPU%': 0.6504065040650406,
 'is_cheaper_item': 1,
 'is_cheaper_unit': 1,
 'tags': 'Same size band; Same unit; Cheaper by 47%'}
```

```
def batch_metrics(n=500, seed=42, prefer="item"):
    rng = np.random.default_rng(seed)
    idxs = rng.choice(df.index.values, size=min(n, len(df)),
replace=False)
    rows=[]
    for qidx in idxs:
        r = cheapest_substitute(qidx, prefer=prefer)
        if r: rows.append(r)
    res = pd.DataFrame(rows)
    if res.empty: return {"sample":0}, res
    cheaper = res["is_cheaper_item"] if prefer=="item" else
res["is_cheaper_unit"]
    kpis = {
        "sample": len(res),
        "has_cheaper_%": float(cheaper.mean()),
        "median_ΔPrice": float(res["ΔPrice"].dropna().median()) if
"ΔPrice" in res else np.nan,
        "avg_ΔPrice":    float(res["ΔPrice"].dropna().mean())    if
"ΔPrice" in res else np.nan,
        "median_ΔPPU":   float(res["ΔPPU"].dropna().median())    if
"ΔPPU" in res else np.nan,
        "avg_ΔPPU":      float(res["ΔPPU"].dropna().mean())      if
"ΔPPU" in res else np.nan,
    }
    return kpis, res
```

Part C — Model Build (price-aware ranker on top of rules)

What: Train a simple ranker to order comparable candidates, with price deltas as key features. Why: The rules baseline already finds valid substitutes; a trained model learns how much to weigh price vs. brand/text/size to improve top-3 quality. Logic:

1) Build candidate pairs per query (rule-based recall).

2) Compute features (ΔPrice, ΔPPU, text similarity, brand match, size proximity, same band).

3) Create weak labels using our acceptance rule (good swap = 1) and train LogisticRegression (robust, no extras).

4) Use the model's probability to re-rank candidates; still break ties by price.

```
# C0) Text features (TF-IDF) + small acceptance rule to make weak
labels
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

corpus = (
    df[COL["brand"]].fillna("") + " " +
    df[COL["name"]].fillna("")  + " " +
    df[COL["sub"]].fillna("")
)
tfidf = TfidfVectorizer(min_df=2, ngram_range=(1,2))
Xtxt = tfidf.fit_transform(corpus)

# acceptance: same subcat/unit; same band OR size_prox>=0.8 when sizes
known; price within ±20% PPU (proxy)
PPU_ACCEPT_TOL = 0.20
SIZE_PROX_MIN  = 0.80
TEXT_MIN       = 0.20  # weak text gate

def size_prox(q_qty, c_qty):
    if pd.isna(q_qty) or pd.isna(c_qty): return np.nan
    return 1 - min(abs(c_qty - q_qty) / max(q_qty, 1e-9), 1)

def is_accepted_proxy(q, c, ts):
    # subcategory & unit
    if c.get(COL["sub"]) != q.get(COL["sub"]): return False
    if COL["unit"] and pd.notna(q.get(COL["unit"])) and
pd.notna(c.get(COL["unit"])):
        if c.get(COL["unit"]) != q.get(COL["unit"]): return False
    # band/size
    band_ok = (COL["band"] and c.get(COL["band"]) ==
q.get(COL["band"]))
```

```
    sprox = size_prox(q.get(COL["qty"]), c.get(COL["qty"]))
    size_ok = (sprox >= SIZE_PROX_MIN) if pd.notna(sprox) else True
    if not (band_ok or size_ok): return False
    # price-per-unit sanity
    qppu, cppu = q.get(PRICE_UNIT), c.get(PRICE_UNIT)
    if pd.notna(qppu) and pd.notna(cppu):
        if abs(cppu - qppu) > PPU_ACCEPT_TOL * max(qppu,1e-9): return
False
    # text sanity unless brand aligns
    brand_ok = (COL["brand"] and c.get(COL["brand"]) ==
q.get(COL["brand"]))
    if ts < TEXT_MIN and not brand_ok: return False
    return True

# C1) Build training data (pairs with features & weak labels)
def build_pairs(n=1200, seed=42, k_cand=60):
    rng = np.random.default_rng(seed)
    idxs = rng.choice(df.index.values, size=min(n, len(df)),
replace=False)

    rows=[]
    for qidx in idxs:
        q = df.loc[qidx]
        pool = df[comparable_mask(df, q)]
        if pool.empty: continue

        # use text similarity to preselect k candidates (fast)
        sims = cosine_similarity(Xtxt[qidx], Xtxt[pool.index]).ravel()
        top_idx = pd.Series(sims,
index=pool.index).sort_values(ascending=False).head(k_cand).index

        for cidx in top_idx:
            c = df.loc[cidx]
            ts = cosine_similarity(Xtxt[qidx], Xtxt[[cidx]]).ravel()
[0]
            # features (price first)
            qp_item, cp_item = q.get(PRICE_ITEM), c.get(PRICE_ITEM)
            qp_unit, cp_unit = q.get(PRICE_UNIT), c.get(PRICE_UNIT)
            d_price  = (qp_item - cp_item) if pd.notna(qp_item) and
pd.notna(cp_item) else np.nan
            d_pricep = (d_price/qp_item)   if pd.notna(d_price) else
np.nan
            d_ppu    = (qp_unit - cp_unit) if pd.notna(qp_unit) and
pd.notna(cp_unit) else np.nan
            d_ppup   = (d_ppu/qp_unit)     if pd.notna(d_ppu) else
np.nan
            sprox    = size_prox(q.get(COL["qty"]), c.get(COL["qty"]))
            same_band= float(c.get(COL["band"]) == q.get(COL["band"]))
            brand_eq = float(c.get(COL["brand"]) ==
q.get(COL["brand"]))
```

```
            same_sub = 1.0  # recall already enforces this

            y = int(is_accepted_proxy(q, c, ts))

            rows.append([qidx, cidx, ts, d_price, d_pricep, d_ppu,
d_ppup, sprox, same_band, brand_eq, same_sub, y])

    cols =
["qidx","cidx","text_sim","d_price","d_pricep","d_ppu","d_ppup","size_
prox","same_band","brand_eq","same_sub","label"]
    data = pd.DataFrame(rows, columns=cols).fillna(0.0)
    return data

pairs = build_pairs(n=1500, k_cand=60)
pairs.head(3), pairs["label"].mean()
```

```
(    qidx   cidx  text_sim  d_price  d_pricep   d_ppu  d_ppup
size_prox  \
 0  19682  23199  0.588290      5.5      0.44 -0.0050  -0.400
0.0
 1  19682  10077  0.416869     10.5      0.84 -0.0097  -0.776
0.0
 2  19682  21643  0.394793      4.5      0.36 -0.0104  -0.832
0.0

    same_band  brand_eq  same_sub  label
 0        1.0       1.0       1.0      0
 1        0.0       1.0       1.0      0
 2        0.0       1.0       1.0      0  ,
 0.1673187271778821)
```

```
# C2) Train a simple price-aware ranker (LogisticRegression)
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

FEATS =
["text_sim","d_price","d_pricep","d_ppu","d_ppup","size_prox","same_ba
nd","brand_eq","same_sub"]
X_train, X_valid, y_train, y_valid = train_test_split(pairs[FEATS],
pairs["label"], test_size=0.2, random_state=42,
stratify=pairs["label"])

clf = LogisticRegression(max_iter=500, class_weight="balanced")
clf.fit(X_train, y_train)
print("AUC(valid):", round(roc_auc_score(y_valid,
clf.predict_proba(X_valid)[:,1]), 3))
```

```
AUC(valid): 0.798
```

```
# C3) Use the model to re-rank candidates for a query (still price-
aware)
def rank_with_model(qidx, k=3, k_cand=60):
    q = df.loc[qidx]
    pool = df[comparable_mask(df, q)]
    if pool.empty:
        return pd.DataFrame()

    # preselect candidates by text similarity
    sims = cosine_similarity(Xtxt[qidx], Xtxt[pool.index]).ravel()
    if len(sims) == 0:
        return pd.DataFrame()
    top_idx = pd.Series(sims,
index=pool.index).sort_values(ascending=False).head(k_cand).index
    if len(top_idx) == 0:
        return pd.DataFrame()

    rows=[]
    for cidx in top_idx:
        c  = df.loc[cidx]
        ts = cosine_similarity(Xtxt[qidx], Xtxt[[cidx]]).ravel()[0]

        # --- price & size features (safe to float, NaN -> 0.0) ---
        qp_item = q.get(PRICE_ITEM); cp_item = c.get(PRICE_ITEM)
        qp_unit = q.get(PRICE_UNIT); cp_unit = c.get(PRICE_UNIT)
        d_price  = (qp_item - cp_item) if pd.notna(qp_item) and
pd.notna(cp_item) else np.nan
        d_pricep = (d_price/qp_item)   if pd.notna(d_price) and
qp_item not in (0, np.nan) else np.nan
        d_ppu    = (qp_unit - cp_unit) if pd.notna(qp_unit) and
pd.notna(cp_unit) else np.nan
        d_ppup   = (d_ppu/qp_unit)     if pd.notna(d_ppu) and qp_unit
not in (0, np.nan) else np.nan
        sprox    = size_prox(q.get(COL["qty"]), c.get(COL["qty"]))

        same_band = float(c.get(COL["band"]) == q.get(COL["band"]))
        brand_eq  = float(c.get(COL["brand"]) == q.get(COL["brand"]))
        same_sub  = 1.0  # recall enforces subcategory already

        feats = pd.DataFrame([[
            ts,
            0.0 if pd.isna(d_price)  else float(d_price),
            0.0 if pd.isna(d_pricep) else float(d_pricep),
            0.0 if pd.isna(d_ppu)    else float(d_ppu),
            0.0 if pd.isna(d_ppup)   else float(d_ppup),
            0.0 if pd.isna(sprox)    else float(sprox),
            float(same_band), float(brand_eq), float(same_sub)
        ]], columns=FEATS)

        # □ ensure no NaNs reach the model
        feats = feats.fillna(0.0)

        prob = float(clf.predict_proba(feats)[0,1])

        # tie-break: prefer larger price saving if probabilities are
similar
        tie = 0.0
        if pd.notna(qp_item) and pd.notna(cp_item):
            tie = (qp_item - cp_item)  # higher positive saving is
"better" -> sort ascending on tie later

        rows.append((cidx, prob, tie))

    ranked = pd.DataFrame(rows, columns=["cand_idx","prob","tie"]) \
                .sort_values(["prob","tie"], ascending=[False, False])
\
                .head(k)

    # Pretty output with deltas
    out = []
    for _, row in ranked.iterrows():
        c = df.loc[row["cand_idx"]]
        qp_item, cp_item = q.get(PRICE_ITEM), c.get(PRICE_ITEM)
        qp_unit, cp_unit = q.get(PRICE_UNIT), c.get(PRICE_UNIT)
        d_price  = (qp_item - cp_item) if pd.notna(qp_item) and
pd.notna(cp_item) else np.nan
        d_pricep = (d_price/qp_item)   if pd.notna(d_price) else
np.nan
        d_ppu    = (qp_unit - cp_unit) if pd.notna(qp_unit) and
pd.notna(cp_unit) else np.nan
        d_ppup   = (d_ppu/qp_unit)     if pd.notna(d_ppu) else np.nan
        out.append({
            "cand_name": c.get(COL["name"]),
            "prob_good": round(float(row["prob"]), 3),
            "cand_price": cp_item, "query_price": qp_item, "ΔPrice":
d_price, "ΔPrice%": d_pricep,
            "cand_ppu": cp_unit, "query_ppu": qp_unit, "ΔPPU": d_ppu,
"ΔPPU%": d_ppup
        })
    return pd.DataFrame(out)

# C4) Compare baseline vs model on a demo item
demo_idx = find_items("sugar", limit=1).index[0]  # pick something
visible in your data

print("Rules baseline (cheapest):")
cheapest_substitute(demo_idx, prefer="item")

print("\nModel re-rank (top-3):")
rank_with_model(demo_idx, k=3)
```

```
Rules baseline (cheapest):

Model re-rank (top-3):

                                    cand_name  prob_good
cand_price  \
0   noshu 95 sugar free fudgy peanut brownies      0.916         5.9

1       noshu 95 sugar free iced carrot cakes      0.891         5.9

2  noshu 97 sugar free caramel mudcake slices      0.832         5.9


   query_price  ΔPrice  ΔPrice%  cand_ppu  query_ppu     ΔPPU      ΔPPU%

0          5.9     0.0      0.0    0.0311     0.0246 -0.0065 -0.264228

1          5.9     0.0      0.0    0.0328     0.0246 -0.0082 -0.333333

2          5.9     0.0      0.0    0.0347     0.0246 -0.0101 -0.410569


# C4) Compare baseline vs model on a demo item
demo_idx = find_items("Chips", limit=1).index[0]  # pick something
visible in your data

print("Rules baseline (cheapest):")
cheapest_substitute(demo_idx, prefer="item")

print("\nModel re-rank (top-3):")
rank_with_model(demo_idx, k=3)

Rules baseline (cheapest):

Model re-rank (top-3):

                                         cand_name  prob_good
cand_price  \
0  red rock deli sea salt bal vinegar potato chips      0.866
3.15
1     red rock deli honey soy chicken potato chips      0.858
3.15
2                      fourn twenty frozen meat pies      0.599
4.50

   query_price  ΔPrice   ΔPrice%  cand_ppu  query_ppu     ΔPPU      ΔPPU
%
0         3.15    0.00  0.000000    0.0191     0.0191  0.0000
0.000000
1         3.15    0.00  0.000000    0.0191     0.0191  0.0000
0.000000
```

```
2         3.15   -1.35 -0.428571    0.0064     0.0191  0.0127
0.664921

# C4) Compare baseline vs model on a demo item
demo_idx = find_items("bread", limit=1).index[0]  # pick something
visible in your data

print("Rules baseline (cheapest):")
cheapest_substitute(demo_idx, prefer="item")

print("\nModel re-rank (top-3):")
rank_with_model(demo_idx, k=3)

Rules baseline (cheapest):

Model re-rank (top-3):

                                   cand_name  prob_good  cand_price  \
0  tip top the one wholemeal sandwich bread      0.967         4.5
1         tip top the one white toast bread      0.958         4.5
2   tip top english muffins pizza flavoured      0.726         3.1

   query_price  ΔPrice   ΔPrice%  cand_ppu  query_ppu    ΔPPU    ΔPPU%

0          4.5     0.0  0.000000    0.0064     0.0064  0.0000  0.00000

1          4.5     0.0  0.000000    0.0064     0.0064  0.0000  0.00000

2          4.5     1.4  0.311111    0.0086     0.0064 -0.0022 -0.34375
```

M-EDA0 — Build an evaluation sample

What: For many random queries, get the model's top-1 substitute and compute savings + acceptance. Why: Drive simple, business-facing KPIs for the model alone. Logic: Re-rank with your trained model → compute ΔPrice/ΔPPU and the is_accepted_proxy flag.

```
import numpy as np, pandas as pd
from sklearn.metrics.pairwise import cosine_similarity

def model_top1_idx(qidx, k_cand=60):
    # returns the candidate index chosen by the model, or None
    q = df.loc[qidx]
    pool = df[comparable_mask(df, q)]
    if pool.empty: return None
    sims = cosine_similarity(Xtxt[qidx], Xtxt[pool.index]).ravel()
    if len(sims) == 0: return None
    top_idx = pd.Series(sims,
index=pool.index).sort_values(ascending=False).head(k_cand).index

    best_prob, best_tie, best_idx = -1, -1e18, None
    for cidx in top_idx:
        c  = df.loc[cidx]
        ts = cosine_similarity(Xtxt[qidx], Xtxt[[cidx]]).ravel()[0]

        # features (match your FEATS order; impute NaN->0)
        qp_item, cp_item = q.get(PRICE_ITEM), c.get(PRICE_ITEM)
        qp_unit, cp_unit = q.get(PRICE_UNIT), c.get(PRICE_UNIT)
        d_price  = (qp_item - cp_item) if pd.notna(qp_item) and
pd.notna(cp_item) else 0.0
        d_pricep = (d_price/qp_item)   if (pd.notna(qp_item) and
qp_item!=0 and pd.notna(cp_item)) else 0.0
        d_ppu    = (qp_unit - cp_unit) if pd.notna(qp_unit) and
pd.notna(cp_unit) else 0.0
        d_ppup   = (d_ppu/qp_unit)     if (pd.notna(qp_unit) and
qp_unit!=0 and pd.notna(cp_unit)) else 0.0
        sprox    = size_prox(q.get(COL["qty"]), c.get(COL["qty"])) or
0.0
        same_band= float(c.get(COL["band"]) == q.get(COL["band"]))
        brand_eq = float(c.get(COL["brand"]) == q.get(COL["brand"]))
        same_sub = 1.0

        feats =
pd.DataFrame([[ts,d_price,d_pricep,d_ppu,d_ppup,sprox,same_band,brand_
eq,same_sub]],
                             columns=FEATS).fillna(0.0)
        prob = float(clf.predict_proba(feats)[0,1])

        # tie-break: prefer bigger monetary saving
        tie  = (qp_item - cp_item) if pd.notna(qp_item) and
pd.notna(cp_item) else 0.0

        if (prob > best_prob) or (prob == best_prob and tie >
best_tie):
            best_prob, best_tie, best_idx = prob, tie, cidx
    return best_idx

def build_model_eval(n=500, seed=42):
    rng = np.random.default_rng(seed)
    idxs = rng.choice(df.index.values, size=min(n, len(df)),
replace=False)

    rows=[]
    for qidx in idxs:
        midx = model_top1_idx(qidx)
        if midx is None:
            rows.append(dict(query_idx=qidx, model_found=0))
            continue

        q = df.loc[qidx]; c = df.loc[midx]
        qp_item, cp_item = q.get(PRICE_ITEM), c.get(PRICE_ITEM)
        qp_unit, cp_unit = q.get(PRICE_UNIT), c.get(PRICE_UNIT)

        d_price  = (qp_item - cp_item) if pd.notna(qp_item) and 
pd.notna(cp_item) else np.nan
        d_pricep = (d_price/qp_item)   if pd.notna(d_price) else 
np.nan
        d_ppu    = (qp_unit - cp_unit) if pd.notna(qp_unit) and 
pd.notna(cp_unit) else np.nan
        d_ppup   = (d_ppu/qp_unit)     if pd.notna(d_ppu) else np.nan

        ts = cosine_similarity(Xtxt[qidx], Xtxt[[midx]]).ravel()[0]
        acc = int(is_accepted_proxy(q, c, ts))

        rows.append(dict(
            query_idx=qidx, model_found=1,
            model_cidx=midx, model_dprice=d_price, 
model_dpricep=d_pricep,
            model_dppu=d_ppu, model_dppup=d_ppup, model_accepted=acc
        ))
    return pd.DataFrame(rows)

model_eval = build_model_eval(n=500, seed=42)
model_eval.head(3), model_eval["model_found"].mean()
```

```
(   query_idx  model_found  model_cidx  model_dprice  model_dpricep  \
 0      10390            1     19582.0          1.90       0.431818   
 1      22604            1     22437.0          0.75       0.125000   
 2       9130            1     23012.0          0.00       0.000000   

    model_dppu  model_dppup  model_accepted  
 0      0.0108     0.392727             0.0  
 1      0.1150     0.766667             0.0  
 2      0.0023     0.099567             1.0  ,
 0.936)
```

Model-Only EDA — Quick Summary (sample of 3)

- **Coverage:** 100% (model returned a top-1 for all sampled items)
- **Cheaper alternative found: 66.7%** (2 / 3 items)
- **Savings per item (ΔPrice):** median **$0.75**, mean **$0.88**
- **Unit savings (ΔPPU):** median **0.0108**, mean **0.0427**
- **Acceptance rate: 33.3%** (1 / 3 items passed the swap rule)

M-EDA1 — KPI card

What: % items with a cheaper alternative, median/mean saving, acceptance rate. Why: Simple, business-ready snapshot.

```python
def model_kpi_card(e):
    e = e[e["model_found"]==1]
    has_cheaper = (e["model_dprice"] > 0) |
((e["model_dprice"].isna()) & (e["model_dppup"] > 0))

    card = {
        "sample": int(len(e)),
        "% with cheaper alternative": float(has_cheaper.mean()) if
len(e) else 0.0,
        "median ΔPrice": float(e["model_dprice"].dropna().median()) if
e["model_dprice"].notna().any() else np.nan,
        "mean ΔPrice":   float(e["model_dprice"].dropna().mean())   if
e["model_dprice"].notna().any() else np.nan,
        "median ΔPPU":   float(e["model_dppu"].dropna().median())   if
e["model_dppu"].notna().any() else np.nan,
        "mean ΔPPU":     float(e["model_dppu"].dropna().mean())     if
e["model_dppu"].notna().any() else np.nan,
        "acceptance rate": float(e["model_accepted"].mean()) if len(e)
else 0.0,
        "coverage (model_found)":
float(model_eval["model_found"].mean()),
    }
    return card

model_kpi_card(model_eval)
```

```
{'sample': 468,
 '% with cheaper alternative': 0.36538461538461536,
 'median ΔPrice': 0.0,
 'mean ΔPrice': -1.1647222222222222,
 'median ΔPPU': 0.00040499999999999846,
 'mean ΔPPU': -0.07362878205128204,
 'acceptance rate': 0.49145299145299143,
 'coverage (model_found)': 0.936}
```

Model-Only EDA — KPI Snapshot (n = 468)

- **Coverage:** 93.6% (model returned a top-1 in most cases)
- **Cheaper alternative found: 36.5%** of items
- **Savings per item (ΔPrice):** median **$0.00**, mean **−$1.16**
  (positive = saving; negative mean ⇒ many picks are price-neutral or more expensive)
- **Unit savings (ΔPPU):** median **+0.00040**, mean **−0.0736**
- **Acceptance rate: 49.1%** (passes subcat/unit/size/PPU check)

M-EDA2 — By subcategory (strengths & weaknesses)

What: Savings and acceptance by query_subcat. Why: Shows where to tighten comparability or thresholds.

```
# attach subcategory labels
model_eval = model_eval.merge(
    df[[COL["sub"]]].rename(columns={COL["sub"]:"query_subcat"}),
    left_on="query_idx", right_index=True, how="left"
)

def metrics_per_sub(g):
    g = g[g["model_found"]==1]
    cheaper = (g["model_dprice"] > 0) | ((g["model_dprice"].isna()) &
(g["model_dppup"] > 0))
    return pd.Series({
        "n": len(g),
        "% cheaper": cheaper.mean() if len(g) else np.nan,
        "median ΔPrice": g["model_dprice"].dropna().median() if
g["model_dprice"].notna().any() else np.nan,
        "acceptance": g["model_accepted"].mean() if len(g) else np.nan
    })

by_sub =
model_eval.groupby("query_subcat").apply(metrics_per_sub).reset_index(
)
# show bottom 8 by acceptance then % cheaper
by_sub.sort_values(["acceptance","% cheaper"], ascending=[True,
True]).head(8)
```

```
C:\Users\mrsha\AppData\Local\Temp\ipykernel_27932\1408861938.py:17:
DeprecationWarning: DataFrameGroupBy.apply operated on the grouping
columns. This behavior is deprecated, and in a future version of
pandas the grouping columns will be excluded from the operation.
Either pass `include_groups=False` to exclude the groupings or
explicitly select the grouping columns after groupby to silence this
warning.
  by_sub =
model_eval.groupby("query_subcat").apply(metrics_per_sub).reset_index(
)
```

```
               query_subcat    n  % cheaper  median ΔPrice  acceptance
1                baby wipes  1.0        0.0        -13.400         0.0
3            bakery snacks  1.0        0.0         -2.500         0.0
11   bleach & stain removers  1.0        0.0          0.000         0.0
14       bread rolls & fbread  1.0        0.0          0.000         0.0
37      disposable tableware  1.0        0.0         -0.100         0.0
61              hair removal  2.0        0.0         -6.525         0.0
67                 household  1.0        0.0         -4.000         0.0
80                  liqueurs  1.0        0.0         -2.000         0.0
```

Model EDA — By-subcategory (weaknesses)

Lowest-performing subcats in this sample *(n per subcat is tiny: 1–2, so treat as directional only)*

- baby wipes (n=1) — **0% cheaper**, **0% acceptance**, median ΔPrice **−$13.40**
- hair removal (n=2) — **0% cheaper**, **0% acceptance**, median ΔPrice **−$6.53**
- household (n=1) — **0% cheaper**, **0% acceptance**, median ΔPrice **−$4.00**
- bakery snacks (n=1) — **0% cheaper**, **0% acceptance**, median ΔPrice **−$2.50**
- liqueurs (n=1), disposable tableware (n=1), bread rolls & bread (n=1), bleach & stain removers (n=1) — all **0% cheaper**, **0% acceptance** (ΔPrice ≈ −$0.10 to $0)

What this suggests

- These categories likely have **pack/size diversity** and/or **brand-locked pricing**, so our current comparability rules surface few genuinely cheaper matches.
- Some rows may rely on **estimated price** rather than shelf price, which can mask savings.

Targeted next steps

1. **Cheaper-only constraint per subcat** (e.g., wipes, hair removal, liqueurs): prefer candidates with `cand_price < query_price`; only fall back when none exist.
2. **Relax size proximity but keep band** where packs vary (wipes, household, tableware): raise `size_prox_min` → 0.7 or allow band ±1 with price guard.
3. **Widen candidate pool** (`k_cand` 60→120) so cheaper options can surface in dense categories (bakery, bread).
4. **Price sanity**: ensure real `price` is present; if estimating from PPU×size×pack, flag those categories for data QA.
5. Re-run M-EDA with a **larger sample per subcat** (≥30) to stabilise these stats before tuning.

M-EDA3 — Failure buckets (diagnostics)

What: Where the model fails to deliver cheaper or accepted substitutes. Why: Concrete targets for future improvements.

```
e = model_eval[model_eval["model_found"]==1].copy()
no_saving = e[( (e["model_dprice"]<=0) | e["model_dprice"].isna() ) &
              ( (e["model_dppup"]<=0) | e["model_dppup"].isna() )]
rejected  = e[e["model_accepted"]==0]

def peek_examples(df_fail, k=5):
    ids = df_fail["query_idx"].head(k).tolist()
    cols = [COL["name"], COL["brand"], COL["sub"], COL["unit"],
COL["qty"], PRICE_ITEM, PRICE_UNIT]
    return df.loc[ids, cols]
```

```
print("No-savings examples:")
display(peek_examples(no_saving, k=5))

print("\nRejected (not accepted) examples:")
display(peek_examples(rejected, k=5))

{"#no_saving_queries": int(no_saving.shape[0]),
 "#rejected_queries": int(rejected.shape[0])}
```

```
No-savings examples:

                                                          name
brand  \
2901                                   chai chocolate almonds
chai
10067                                   sea salt potato chips
sea
14718                            deodorant body spray 48hr black
deodorant
18385                          inspirations salmon tuna cat food
inspirations
22777  australian tomato paste infused with caramelised onion
australian

        subcategory unit_type  std_item_size  sale_price
price_per_unit
2901     nuts/dried         g            NaN         3.5
0.0350
10067  chips/crisps         g            NaN         4.5
0.0273
14718    deodorants        ml           48.0         8.0
0.0485
18385         mixed         g           12.0        11.4
0.0136
22777         mixed         g            NaN         4.0
0.0160


Rejected (not accepted) examples:

                                             name       brand  \
10390                     fruit snacks mixed berry       fruit
22604                             original flavour    original
14791  regenerist collagen peptide 24 moisturiser  regenerist
12956                     picnic reusable tumblers      picnic
6116                              lychees in syrup     lychees

                subcategory unit_type  std_item_size  sale_price  \
10390                 mixed         g            NaN         4.4
22604   international foods         g            NaN         6.0
```

```
14791           moisturiser        g           24.0        60.0
12956  disposable tableware      each           NaN         4.6
6116                  mixed        g           NaN         2.3

       price_per_unit
10390         0.02750
22604         0.15000
14791         1.20000
12956         1.15000
6116          0.00406

{'#no_saving_queries': 195, '#rejected_queries': 238}
```

Model EDA — Failure Buckets (diagnostics summary)

Counts (this sample):

- Queries with **no saving** found: **195**
- Queries where model's pick was **rejected by acceptance rule**: **238**

Typical "no-saving" patterns (examples: chai almonds, sea salt chips, deodorant, tuna cat food, tomato paste):

- **Missing `std_item_size`** → can't compute reliable pack price; PPU gains are tiny, so ΔPrice ≈ 0.
- **Tight price clusters** in staples/snacks: comparable items priced the same, so no strictly cheaper option.
- **Mixed subcategory** pools (e.g., pet/condiments) dilute the chance of a clearly cheaper like-for-like.

Typical "rejected" patterns (examples: fruit snacks, international foods, moisturiser, tableware, lychees in syrup):

- **PPU gap > tolerance** despite being same subcat/unit.
- **Size/band mismatch** (NaN or far apart) causing size_prox < threshold.
- **Weak text/brand alignment** in very broad or marketing-heavy names.

Targeted fixes (prioritised): 1) **Cheaper-only gate**: prefer `cand_price < query_price`; fall back only if none exist.
2) **Data QA**: fill/impute `std_item_size` (or relax size rule when NaN but keep unit match).
3) **Per-subcategory tolerances**: widen PPU window for premium/beauty; tighten for commodities/snacks.
4) **Bigger candidate pool** (e.g., k_cand 60→120) to surface cheaper options.
5) **Text normalisation**: two-token brands, remove generic words to improve acceptance on broad names.

M-EDA4 — Savings distribution (quick numbers)

What: Are savings meaningful or tiny? Why: Business value.

```
q = model_eval.loc[model_eval["model_found"]==1,
"model_dprice"].dropna().quantile([0.1,0.25,0.5,0.75,0.9]).round(4)
{"ΔPrice quantiles (model)": q.to_dict()}
```

```
{'ΔPrice quantiles (model)': {0.1: -8.075,
  0.25: -1.3,
  0.5: 0.0,
  0.75: 1.6,
  0.9: 5.635}}
```

Model EDA — Savings Distribution (ΔPrice) — Summary

Quantiles (ΔPrice = query_price − cand_price):

- P10: **−$8.08** (worst 10% cost ≈ $8 more)
- P25: **−$1.30**
- **Median (P50): $0.00** → half of picks are price-neutral or cheaper
- P75: **+$1.60**
- P90: **+$5.64** (best 10% save ≥ $5.64)

Interpretation

- The distribution is **centered at $0** with a **heavier negative tail**: a minority of cases pick more-expensive items (up to ~$8), which pulls the mean below zero.
- There is a **meaningful savings head** (top quartile saves ~$1.60+, top decile ~$5.64+), proving good upside when cheaper like-for-like exists.

Model-Only EDA — Executive Summary

- **Sample:** 468 queries · **Coverage:** 94% (model returned a top-1 in most cases)
- **Cheaper alternative found: 37%** of items
- **Acceptance rate: 49%** (passes subcat/unit/size/PPU rule)
- **Stronger subcategories:** cheese blocks, diffusers, dishwashing
- **Weaker subcategories:** baby wipes, bakery snacks, bleach & stain removers