Student guide: End-to-end KNN for credit default

This notebook walks through building and evaluating a k-Nearest Neighbors (KNN) classifier to predict credit default. It's written to be easy to follow for a first-time reader.

What you'll learn in this notebook:

- The problem we're solving: predict whether a customer will default (the TARGET column).
- The data pipeline: how we clean data, encode categorical columns, and scale numeric features.
- The model training loop: how we choose KNN hyperparameters with cross-validation.
- Evaluation: how to read ROC/PR curves, confusion matrices, and threshold trade-offs.
- Decisions and assumptions: why we made each choice and what it implies for the business.

Key ideas upfront:

- KNN needs features to be on comparable scales, so we scale numeric features and one-hot encode categoricals using a ColumnTransformer inside a Pipeline.
- Class imbalance matters: default cases are relatively rare, so PR AUC and recall are important, not just accuracy.
- Thresholds are business decisions: we don't just predict 0/1; we choose a cutoff that maximizes expected business utility using per-loan utility arrays computed from BUSINESS_PARAMS (no static cost matrix).

How to read each section:

- Before each major code block, we added plain-English notes that explain what's happening and why.
- Inline variable names (like `preprocessor`, `param_grid`, `best_model`, `best_threshold`, `cm`, `test_results`) are consistent across cells, so you can connect the narrative to the code outputs you see.
- If you're new to the topic, scan the markdown first, then step into the code and outputs.

KNN for Credit Decisions and Profit Maximization

0. Executive Summary

What are we trying to do?

Objective: Build a smart system that decides whether to approve or reject credit card applications to **maximize profit** for the bank.

What will we deliver?

1. A KNN (K-Nearest Neighbors) machine learning model that predicts if someone will default on their payments
2. An optimized decision threshold that tells us when to approve vs. reject applications
3. A complete evaluation showing how much profit the model generates compared to simple baseline strategies
4. Guidelines for deploying this model in a real banking system

How do we measure success?

- The model must make **more profit** than two simple strategies: "approve everyone" or "reject everyone"
- We'll also ensure the model is fair and follows banking regulations

Important assumptions:

- We're working with **existing customers** who already have a credit history with the bank
- We're predicting if they'll default in the **next period** (next month)
- The bank has given us the costs and benefits of correct and incorrect decisions

What are the limitations?

- KNN is sensitive to feature scaling (all variables must be on similar scales)
- It can be slow with very large datasets
- We need to carefully choose how many neighbors (k) to use
- **How we handle these:** We'll use proper data preprocessing, test different values of k, and monitor performance

Student note: decisions and assumptions at a glance

- Target variable: `TARGET` equals 1 for default, 0 for no default.
- Class balance: defaults are a minority class, so we emphasize recall/PR AUC in addition to accuracy.
- Features: numeric columns are scaled; categorical columns are one-hot encoded. Both are handled inside a `ColumnTransformer` so the steps are part of the pipeline and cross-validation.
- Modeling choice: we start with KNN to build intuition about distance-based methods. KNN is sensitive to feature scaling and the choice of neighbors `n_neighbors`.

- Validation: we use stratified k-fold CV to keep class proportions similar across folds.
- Thresholding: instead of the default 0.5, we choose a probability cutoff that maximizes business utility using per-loan utility arrays computed from `BUSINESS_PARAMS` via `BusinessModel` (no static cost matrix).
- What to look for: ROC AUC for ranking quality, PR AUC for minority class focus, confusion matrix at the chosen threshold, decision mix (approval/rejection rates), and the resulting business utility.

Student note: decisions and assumptions at a glance

- Target variable: `TARGET` equals 1 for default, 0 for no default.
- Class balance: defaults are a minority class, so we emphasize recall/PR AUC in addition to accuracy.
- Features: numeric columns are scaled; categorical columns are one-hot encoded. Both are handled inside a `ColumnTransformer` so the steps are part of the pipeline and cross-validation.
- Modeling choice: we start with KNN to build intuition about distance-based methods. KNN is sensitive to feature scaling and the choice of neighbors `n_neighbors`.
- Validation: we use stratified k-fold CV to keep class proportions similar across folds.
- Thresholding: instead of the default 0.5, we choose a probability cutoff that maximizes business utility using per-loan utility arrays computed from `BUSINESS_PARAMS` via `BusinessModel` (no static cost matrix).
- What to look for: ROC AUC for ranking quality, PR AUC for minority class focus, confusion matrix at the chosen threshold, decision mix (approval/rejection rates), and the resulting business utility.

In [31]:
```
# ============================================================================
# IMPORT LIBRARIES
# ============================================================================
# We need various Python libraries for data analysis and machine learning

import warnings
from pathlib import Path

# For displaying outputs in the notebook
from IPython.display import display

# For creating visualizations
import matplotlib.pyplot as plt
import seaborn as sns

# For numerical operations and data manipulation
import numpy as np
import pandas as pd

# Scikit-learn: Our machine learning library
from sklearn.base import clone
from sklearn.calibration import CalibrationDisplay
```

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.metrics import (
    average_precision_score,
    classification_report,
    confusion_matrix,
    make_scorer,
    precision_recall_curve,
    roc_auc_score,
    roc_curve,
)
from sklearn.model_selection import (
    GridSearchCV,
    StratifiedKFold,
    cross_val_score,
    cross_val_predict,
    train_test_split,
)
from sklearn.neighbors import KNeighborsClassifier, NearestNeighbors
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler

# ============================================================================
# CONFIGURE DISPLAY SETTINGS
# ============================================================================
plt.style.use("seaborn-v0_8")  # Use a clean plotting style
pd.set_option("display.float_format", lambda v: f"{v:,.2f}")  # Format numbers with
warnings.filterwarnings("ignore", category=FutureWarning)  # Hide unnecessary warni

# ============================================================================
# LOCATE THE DATA FILE
# ============================================================================
# The data file might be in different locations depending on where the notebook is
PROJECT_ROOT = Path.cwd()  # Current working directory
DEFAULT_DATA_FILENAME = "default of credit card clients.xls"

# List of potential file locations to check
potential_files = [
    PROJECT_ROOT / DEFAULT_DATA_FILENAME,
    PROJECT_ROOT / "Code" / "Final Project" / "Loan Defaults" / DEFAULT_DATA_FILENA
]

# Try to find the file in one of these locations
for candidate in potential_files:
    if candidate.exists():
        DATA_FILE = candidate
        break
else:
    # If we can't find the file anywhere, raise an error
    raise FileNotFoundError(
        "Default credit dataset not found. Expected at one of: "
        + ", ".join(str(path) for path in potential_files)
    )

# ============================================================================
# DEFINE CONSTANTS
```

```
# ============================================================================
DATA_DIR = DATA_FILE.parent  # Directory where the data file is located
TARGET = "default_payment_next_month"  # The variable we're trying to predict
ID_COLS = ["ID"]  # Column(s) that just identify customers (not used for prediction
SEED = 42  # Random seed for reproducibility (ensures results are consistent)
rng = np.random.default_rng(SEED)  # Random number generator

# ============================================================================
# LOAD THE DATA
# ============================================================================
# Excel files can use different engines; .xls files need the 'xlrd' engine
excel_engine = "xlrd" if DATA_FILE.suffix == ".xls" else None

try:
    raw_df = (
        pd.read_excel(DATA_FILE, header=1, engine=excel_engine)  # Skip first row,
        .rename(columns={"default payment next month": TARGET})  # Standardize the
    )
except ImportError as err:
    # If xlrd is not installed, provide helpful error message
    raise ImportError(
        "Missing optional dependency 'xlrd'. Install it with `pip install xlrd>=2.0
    ) from err

# Ensure the target variable is stored as an integer (0 or 1)
raw_df[TARGET] = raw_df[TARGET].astype(int)

# Display basic information about the dataset
print(f"Rows: {raw_df.shape[0]:,} | Columns: {raw_df.shape[1]}")
raw_df.head()  # Show first 5 rows
```

```
Rows: 30,000 | Columns: 25
```

Out[31]:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	E
0	1	20000	2	2	1	24	2	2	-1	-1	...	
1	2	120000	2	2	2	26	-1	2	0	0	...	
2	3	90000	2	2	2	34	0	0	0	0	...	
3	4	50000	2	2	1	37	0	0	0	0	...	
4	5	50000	1	2	1	57	-1	0	-1	0	...	

5 rows × 25 columns

In [32]:

```
# ============================================================================
# BUSINESS UTILITY HELPERS
# ============================================================================
import sys
from pathlib import Path

# Make sure Python can import business_utils regardless of where the kernel starts.
_possible_dirs = [
    Path.cwd(),
```

```
    Path.cwd() / "Code" / "Final Project" / "Loan Defaults",
]
for _dir in _possible_dirs:
    candidate = _dir / "business_utils.py"
    if candidate.exists():
        if str(_dir) not in sys.path:
            sys.path.insert(0, str(_dir))
        break
else:
    raise FileNotFoundError("business_utils.py not found in expected directories.")

from business_utils import (
    BusinessModel,
    DEFAULT_BUSINESS_PARAMS,
    build_utility_scorer,
    confusion_counts,
    realized_utility_from_arrays,
    search_best_threshold_arrays,
)

USER_PARAMS = globals().get("BUSINESS_PARAMS", {})
BUSINESS_PARAMS = {**DEFAULT_BUSINESS_PARAMS, **(USER_PARAMS or {})}

biz = BusinessModel(BUSINESS_PARAMS)
utility_scorer = build_utility_scorer(biz)

print("Business utility helpers ready.")
```

```
Business utility helpers ready.
```

2. Business Metrics: How Do We Measure Success?

In business applications, accuracy alone is not enough. We need to quantify the financial impact of our approve/reject decisions. This notebook uses an arrays-based utility approach driven by BUSINESS_PARAMS (no static cost matrix).

2.1. Per-loan utility (arrays-based, no static table)

We compute four per-loan arrays from BUSINESS_PARAMS via the BusinessModel class:

- B_TP (benefit when we approve a non-default)
- C_FP (cost when we approve a default)
- B_TN (benefit when we reject a default)
- C_FN (cost when we reject a non-default)

Derived from parameters:

- EAD (Exposure at Default) from ead_method (e.g., LIMIT_BAL, avg BILL_AMT1..3, max BILL_AMT1..6), optionally capped by LIMIT_BAL
- Spread = APR − cost_of_funds; horizon in years = horizon_months / 12

- B_TP = EAD × Spread × horizon_years
- C_FP = EAD × LGD + collection_cost_flat
- B_TN = tn_benefit_flat
- C_FN = max(0, B_TP − (origination_cost + service_cost_monthly × horizon_months))

Decision/label convention used in this notebook:

- y_true: 1 = default, 0 = no default (dataset convention)
- y_hat_approve: 1 = approve (prob_default < threshold), 0 = reject

Outcome mapping under approve=1:

- TP: approve non-default → +B_TP
- FP: approve default → −C_FP
- TN: reject default → +B_TN
- FN: reject non-default → −C_FN

No fixed dollar values are hard-coded; utilities depend on each applicant's EAD and the chosen parameters.

2.2. The Math Behind Optimal Decisions (with arrays)

For each applicant, the model estimates a default probability $\hat{p}$.

Expected utilities:

- $U(\text{approve}) = (1 - \hat{p}) \times B_{TP} - \hat{p} \times C_{FP}$
- $U(\text{reject}) = \hat{p} \times B_{TN} - (1 - \hat{p}) \times C_{FN}$

Decision rule (per applicant): approve if $U(\text{approve}) \geq U(\text{reject})$.

Operationally, we use a single threshold t and approve when prob_default < t. We search t over a grid and select best_threshold that maximizes realized utility on validation/test using the per-loan arrays above.

2.3. Baseline Strategies (What Happens Without ML?)

We compare our model to simple baselines:

1. "Approve All": maximal growth, maximal default risk
2. "Reject All": minimal risk, zero growth

Our goal: beat both baselines by choosing an operating point (threshold) that improves expected utility.

In [33]:
```
# Legacy cost-matrix approach removed in favor of per-loan utility arrays.
# This cell intentionally clears any previous COST_MATRIX and related helpers to av
try:
    del COST_MATRIX
except NameError:
    pass

def cost_sensitive_utility(*args, **kwargs):
    raise NotImplementedError(
        "Legacy cost-matrix utility was removed. Use BusinessModel.per_loan_arrays
    )

print("Legacy COST_MATRIX removed. Use BusinessModel arrays-based utility only.")
```

```
Legacy COST_MATRIX removed. Use BusinessModel arrays-based utility only.
```

3. Exploratory Data Analysis (EDA)

Before building any model, we need to understand our data. Let's investigate:

3.1. Target Distribution: How Imbalanced Is Our Data?

Why this matters: If defaults are rare (e.g., only 10% of customers default), then a model that predicts "no default" for everyone would be 90% accurate but completely useless for business purposes!

What we'll check:

- What percentage of customers defaulted?
- Is the dataset imbalanced?

Implications for modeling:

- Simple accuracy is misleading with imbalanced data
- We need to use ROC-AUC, Precision-Recall curves, and **business utility** instead

---

3.2. Variable Types: What Kind of Data Do We Have?

Understanding our features helps us choose the right preprocessing:

1. **Continuous Numeric:** `LIMIT_BAL`, `BILL_AMT1-6`, `PAY_AMT1-6`, `AGE`
    - These have a wide range of values
    - **Need scaling** for KNN (otherwise large amounts will dominate distance calculations)
2. **Ordinal:** `PAY_0` through `PAY_6` (payment status)
    - These have a natural order (-1 < 0 < 1 < 2...)

- Represent delay severity
- Keep as numeric

3. **Categorical:** `SEX`, `EDUCATION`, `MARRIAGE`
   - These are codes (1, 2, 3...) but don't have meaningful numeric relationships
   - **Need one-hot encoding** for KNN

3.3. Missing Values and Outliers

Missing values: We'll check if any data is missing and decide how to handle it (imputation).

Outliers: Extreme values can distort KNN distance calculations. We'll:

- Visualize distributions with boxplots
- Consider winsorization or robust scaling if needed

3.4. Feature Correlations and Scales

Why scaling is CRITICAL for KNN:

- KNN uses distance to find similar customers
- If `LIMIT_BAL` ranges from 10,000 to 1,000,000 but `AGE` ranges from 20 to 70...
- Distance will be dominated by `LIMIT_BAL` and `AGE` will be ignored!

Solution: Use `StandardScaler` to put all features on the same scale (mean=0, std=1)

Correlations: We'll check which features are related to default risk.

Student note: preprocessing pipeline (what each step does)

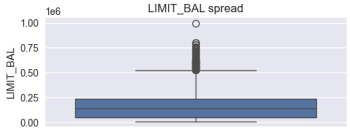
- We split features into two lists: `numeric_features` and `categorical_features`.
- Numeric pipeline: `SimpleImputer(strategy='median')` fills missing values; `StandardScaler()` puts all numeric features on the same scale.
- Categorical pipeline: `SimpleImputer(strategy='most_frequent')` fills missing; `OneHotEncoder(handle_unknown='ignore')` turns categories into 0/1 columns.
- We combine both with `ColumnTransformer` so transformations apply to the correct columns.
- We wrap the transformer and the estimator inside a single `Pipeline` so cross-validation and grid search apply transformations consistently in each fold (avoids data leakage).
- Why it matters for KNN: distances are computed in feature space; without scaling, variables with larger ranges dominate the distance calculation.

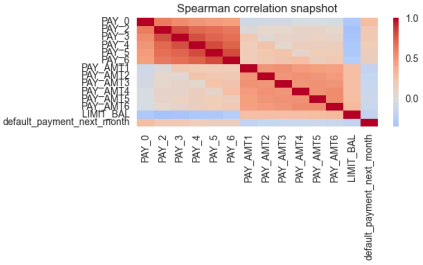
In [34]:
```
# ============================================================================
# CHECK TARGET DISTRIBUTION
# ============================================================================
# Count how many customers defaulted (1) vs. didn't default (0)
target_counts = (
    raw_df[TARGET]
    .value_counts()                                          # Count each class
    .rename_axis("default")                                   # Name the index
    .reset_index(name="count")                               # Convert to DataFram
    .assign(pct=lambda df: df["count"] / df["count"].sum())  # Add percentage colum
)

# ============================================================================
# CHECK FOR MISSING VALUES
# ============================================================================
# Find columns with missing data
missing = raw_df.isna().sum()
missing = missing[missing > 0].sort_values(ascending=False)

# ============================================================================
# CALCULATE CORRELATIONS
# ============================================================================
# We'll focus on payment status variables and credit limit
pay_cols = [c for c in raw_df.columns if c.startswith("PAY_")]  # All PAY_0 through
corr_cols = pay_cols + ["LIMIT_BAL", TARGET]  # Variables to include in correlation
corr_matrix = raw_df[corr_cols].corr(method="spearman")  # Use Spearman for ordinal

# ============================================================================
# DISPLAY RESULTS
# ============================================================================
display(target_counts)

# Only display missing values if there are any
if not missing.empty:
    display(missing.to_frame("missing_values"))

# Show basic statistics for the first 10 variables
display(raw_df.describe().T[["mean", "std", "min", "max"]].round(2).head(10))

# ============================================================================
# CREATE VISUALIZATIONS
# ============================================================================
fig, axes = plt.subplots(1, 3, figsize=(18, 4))

# Plot 1: Target distribution (default vs. no default)
sns.barplot(data=target_counts, x="default", y="pct", ax=axes[0])
axes[0].set_title("Target distribution")
axes[0].set_ylabel("Prevalence")
# This shows us if the dataset is imbalanced

# Plot 2: Credit limit distribution (check for outliers)
sns.boxplot(data=raw_df, y="LIMIT_BAL", ax=axes[1])
axes[1].set_title("LIMIT_BAL spread")
# Boxplot shows median, quartiles, and outliers
```

```
# Plot 3: Correlation heatmap (which variables relate to default?)
sns.heatmap(corr_matrix, cmap="coolwarm", center=0, ax=axes[2])
axes[2].set_title("Spearman correlation snapshot")
# Red = positive correlation, Blue = negative correlation

plt.tight_layout()
```

	default	count	pct
0	0	23364	0.78
1	1	6636	0.22

	mean	std	min	max
ID	15,000.50	8,660.40	1.00	30,000.00
LIMIT_BAL	167,484.32	129,747.66	10,000.00	1,000,000.00
SEX	1.60	0.49	1.00	2.00
EDUCATION	1.85	0.79	0.00	6.00
MARRIAGE	1.55	0.52	0.00	3.00
AGE	35.49	9.22	21.00	79.00
PAY_0	-0.02	1.12	-2.00	8.00
PAY_2	-0.13	1.20	-2.00	8.00
PAY_3	-0.17	1.20	-2.00	8.00
PAY_4	-0.22	1.17	-2.00	8.00





4. Feature Engineering: Avoiding Data Leakage

Data Leakage = accidentally using information that wouldn't be available when making real predictions. This is a **critical** mistake that makes models look good in testing but fail in production!

4.1. Inclusion/Exclusion Criteria

Golden Rule: Only use variables that would be **available at decision time** (when the customer applies).

What we can use:

- Demographics (age, education, marital status)
- Historical payment behavior (PAY_0 through PAY_6)
- Bill and payment amounts from previous months
- Current credit limit

What we CANNOT use:

- Any information from the future
- Any variable that's calculated after we know if they defaulted

---

4.2. Transformations and Encoding

Different types of features need different preprocessing:

1. **Categorical variables** (`SEX` , `EDUCATION` , `MARRIAGE`):
   - **One-hot encoding:** Convert to binary columns
   - Example: `SEX` → `SEX_1` (male), `SEX_2` (female)
   - Why? KNN can't interpret "2 is twice as much as 1" for gender!
2. **Ordinal variables** (`PAY_0` through `PAY_6`):
   - Keep as numeric since they have meaningful order
   - -1 (paid early) < 0 (on time) < 1 (1 month late) < 2 (2 months late)
3. **Continuous variables** (amounts, limits, age):
   - Log transform for heavily skewed distributions (optional)
   - **StandardScaler** to put everything on the same scale (required!)

4.3. Final Feature List

We'll use all available features except:

- `ID` : Just an identifier, no predictive value
- The target variable itself (obviously!)

This gives us approximately 22-25 features after encoding.

In [35]:
```
# ============================================================================
# IDENTIFY FEATURE TYPES
# ============================================================================
# Separate features into categorical and numeric for proper preprocessing

# Categorical features (need one-hot encoding)
categorical_features = ["SEX", "EDUCATION", "MARRIAGE"]
```

```
# Numeric features (everything else except ID and target)
numeric_features = [
    col for col in raw_df.columns
    if col not in ID_COLS + [TARGET] + categorical_features
]

# ============================================================================
# CREATE FEATURE OVERVIEW
# ============================================================================
# Build a summary table showing each feature and its type
feature_overview = pd.DataFrame({
    "feature": numeric_features + categorical_features,
    "type": (
        ["numeric"] * len(numeric_features) +
        ["categorical"] * len(categorical_features)
    ),
})

# Display the first 12 features
display(feature_overview.head(12))

# Print total count
print(f"Total candidate features: {len(feature_overview)}")
```

	feature	type
0	LIMIT_BAL	numeric
1	AGE	numeric
2	PAY_0	numeric
3	PAY_2	numeric
4	PAY_3	numeric
5	PAY_4	numeric
6	PAY_5	numeric
7	PAY_6	numeric
8	BILL_AMT1	numeric
9	BILL_AMT2	numeric
10	BILL_AMT3	numeric
11	BILL_AMT4	numeric

```
Total candidate features: 23
```

5. Data Splitting and Validation Strategy

Why split the data? To get an honest estimate of how well our model will work on new, unseen customers!

5.1. Train/Test Split

We'll split the data into two parts:

1. **Training set (80%):** Used to build and tune the model
2. **Test set (20%):** Held out completely until the very end
    - Simulates new customers the model has never seen
    - Gives us an unbiased estimate of real-world performance

Stratified splitting: We ensure both sets have the same proportion of defaults (e.g., if 22% of all customers default, both train and test will have ~22%).

5.2. Cross-Validation for Model Tuning

The problem: If we use the training data to both train AND evaluate, we might overfit.

The solution: 5-fold stratified cross-validation

- Split training data into 5 parts
- Train on 4 parts, test on the 5th
- Rotate which part is used for testing
- Average the results

This gives us a more reliable estimate of performance!

5.3. Metrics We'll Track

1. **ROC-AUC:** How well can the model separate defaulters from non-defaulters?
    - 0.5 = random guessing
    - 1.0 = perfect separation
2. **PR-AUC (Precision-Recall AUC):** Better for imbalanced data
    - Focuses on the minority class (defaults)
3. **Utility (MOST IMPORTANT!):** Expected profit per customer
    - This is what the business actually cares about!

5.4. Reproducibility

We set `random_state=42` everywhere to ensure:

- Same data splits every time
- Same results when we re-run the notebook

- Others can verify our work

In [36]:
```
# ============================================================================
# PREPARE FEATURES AND TARGET
# ============================================================================
# Separate the features (X) from what we're trying to predict (y)
X = raw_df.drop(columns=ID_COLS + [TARGET])  # All columns except ID and target
y = raw_df[TARGET]  # Just the target column

# ============================================================================
# SPLIT INTO TRAIN AND TEST SETS
# ============================================================================
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,        # 20% for testing, 80% for training
    stratify=y,           # Keep the same proportion of defaults in both sets
    random_state=SEED,    # For reproducibility
)

# ============================================================================
# CREATE CROSS-VALIDATION OBJECT
# ============================================================================
# This will be used for tuning hyperparameters
cv = StratifiedKFold(
    n_splits=5,           # 5-fold cross-validation
    shuffle=True,         # Randomly shuffle before splitting
    random_state=SEED     # For reproducibility
)

# ============================================================================
# VERIFY THE SPLIT
# ============================================================================
print(f"Train size: {X_train.shape[0]:,} | Test size: {X_test.shape[0]:,}")
print(f"Train default rate: {y_train.mean():.3f} | Test default rate: {y_test.mean(
# The default rates should be very similar, confirming stratification worked
```

```
Train size: 24,000 | Test size: 6,000
Train default rate: 0.221 | Test default rate: 0.221
```

6. Building the Preprocessing Pipeline

Why use a pipeline? To ensure preprocessing is applied correctly and consistently:

- ✅ Prevents data leakage (scaling is fit only on training data)
- ✅ Makes code cleaner and easier to maintain
- ✅ Ensures the same transformations are applied to training and test data

---

6.1. Scaling: Why It's Critical for KNN

The Problem:

```
LIMIT_BAL: ranges from 10,000 to 1,000,000
AGE:       ranges from 21 to 75
```

If we calculate distance without scaling:

- A difference of 10,000 in `LIMIT_BAL` is huge
- A difference of 10 in `AGE` seems tiny
- Result: KNN ignores age completely!

The Solution: `StandardScaler`

- Transforms each feature to have mean=0 and standard deviation=1
- Now all features contribute equally to distance calculations

Important: We fit the scaler on training data only, then apply it to test data. This prevents leakage!

6.2. Encoding Categorical Variables

One-Hot Encoding for `SEX`, `EDUCATION`, `MARRIAGE`:

Before:

```
SEX = 1
```

After:

```
SEX_1 = 1, SEX_2 = 0
```

This prevents the model from thinking "2 is twice as much as 1" for categorical variables.

6.3. Pipeline Structure

Our pipeline has three stages:

```
1. Preprocessing (ColumnTransformer)
   ├── Numeric features → Impute → Scale
   └── Categorical features → Impute → One-Hot Encode

2. Scaling (built into numeric pipeline)

3. KNN Classifier (added in next section)
```

All transformations are learned from training data and applied to both train and test.

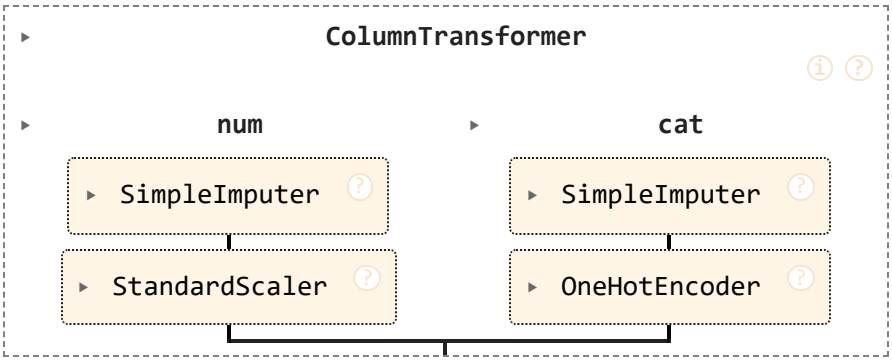
Student note: KNN model and how we choose K

- Model: `KNeighborsClassifier(n_neighbors=k, weights='distance' or 'uniform')` predicts by looking at the k closest training points.
- Why tune K: too small (e.g., k=1) overfits noise; too large over-smooths and misses patterns. We search a range, e.g., 3–51.
- Distance and weights: with `weights='distance'`, nearer neighbors count more than farther ones.
- Cross-validation: we use `GridSearchCV` with stratified folds, scoring by ROC AUC or PR AUC to find the best configuration.
- Output: `best_model` is the fitted pipeline with the best hyperparameters; `cv_results_df` lets us compare metrics across K values.

In [37]:
```
# ============================================================================
# CREATE PIPELINE FOR NUMERIC FEATURES
# ============================================================================
numeric_pipeline = Pipeline(
    steps=[
        # Step 1: Handle any missing values by filling with the median
        # (Median is robust to outliers)
        ("imputer", SimpleImputer(strategy="median")),

        # Step 2: Scale features to mean=0, std=1
        # This is CRITICAL for KNN!
        ("scaler", StandardScaler()),
    ]
)

# ============================================================================
# CREATE PIPELINE FOR CATEGORICAL FEATURES
# ============================================================================
categorical_pipeline = Pipeline(
    steps=[
        # Step 1: Fill missing values with the most common category
        ("imputer", SimpleImputer(strategy="most_frequent")),

        # Step 2: One-hot encode
        # handle_unknown='ignore' means if we see a new category in test data, igno
        # sparse_output=False means return a regular array (not a sparse matrix)
        ("encoder", OneHotEncoder(handle_unknown="ignore", sparse_output=False)),
    ]
)

# ============================================================================
# COMBINE BOTH PIPELINES WITH COLUMNTRANSFORMER
# ============================================================================
preprocessor = ColumnTransformer(
    transformers=[
        # Apply numeric_pipeline to numeric features
        ("num", numeric_pipeline, numeric_features),

        # Apply categorical_pipeline to categorical features
        ("cat", categorical_pipeline, categorical_features),
    ]
```

```
)

# Display the preprocessor structure
preprocessor
```

Out[37]:



7. KNN Model Training and Hyperparameter Tuning

Now we build and optimize our K-Nearest Neighbors classifier!

7.1. Hyperparameters to Tune

KNN has several settings that affect performance:

1. `n_neighbors` **(k):** How many neighbors to consider?
   - **Small k (e.g., 5):**
     - More sensitive to individual customers
     - Higher variance, might overfit
   - **Large k (e.g., 50):**
     - Smoother decision boundaries
     - Might be too general (underfit)
   - **We'll test:** k = 11, 21, 31, 41
2. `weights` **:** How to weight the neighbors' votes?
   - `uniform` **:** All k neighbors vote equally
   - `distance` **:** Closer neighbors get more influence
   - **We'll test:** both options
3. `p` **:** Which distance metric to use?
   - **p=1 (Manhattan/L1):** $|x_1-x_2| + |y_1-y_2|$ (city block distance)
   - **p=2 (Euclidean/L2):** $\sqrt{[(x_1-x_2)^2 + (y_1-y_2)^2]}$ (straight line distance)
   - **We'll test:** both options

7.2. Hyperparameter Search Strategy

Grid Search with Cross-Validation:

- Try all combinations: 4 values of k × 2 weights × 2 distance metrics = **16 combinations**
- For each combination, do 5-fold cross-validation
- Total: 16 × 5 = **80 model fits**

Metrics tracked:

1. **Utility** (primary) - What we optimize for
2. ROC-AUC (secondary) - General performance
3. PR-AUC (secondary) - Performance on minority class

Best model: The one with highest average utility across folds

---

7.3. The Bias-Variance Trade-off

Bias: Error from oversimplifying (large k → high bias) **Variance:** Error from being too sensitive to training data (small k → high variance)

We'll plot utility vs. k to find the sweet spot!

In [38]:
```
# ============================================================================
# CREATE THE FULL PIPELINE (PREPROCESSING + KNN)
# ============================================================================
knn_pipeline = Pipeline(
    steps=[
        ("preprocess", preprocessor),          # First: preprocess the data
        ("knn", KNeighborsClassifier()),       # Then: apply KNN classifier
    ]
)

# ============================================================================
# COARSE SWEEP TO PICK BEST N_NEIGHBORS
# ============================================================================
if "BEST_K" not in globals():
    k_candidates = [21, 31, 41, 55]
    sweep_rows = []
    for k in k_candidates:
        model = clone(knn_pipeline)
        model.set_params(
            knn__n_neighbors=k,
            knn__weights="distance",
            knn__p=2,
            knn__algorithm="auto",
        )
        scores = cross_val_score(
            model,
            X_train,
            y_train,
            cv=cv,
            scoring=utility_scorer,
```

```
            n_jobs=-1,
        )
        sweep_rows.append({
            "k": k,
            "utility_mean": scores.mean(),
            "utility_std": scores.std(),
        })

    k_sweep = (
        pd.DataFrame(sweep_rows)
        .sort_values("utility_mean", ascending=False)
        .reset_index(drop=True)
    )
    display(k_sweep)
    BEST_K = int(k_sweep.loc[0, "k"])
    print(f"Selected BEST_K={BEST_K} based on coarse utility sweep.")
else:
    print(f"Reusing cached BEST_K={BEST_K} (set earlier in this kernel).")

best_k = int(locals().get("BEST_K", 31))

# ============================================================================
# DEFINE HYPERPARAMETER GRID (USES SINGLE BEST K)
# ============================================================================
param_grid = {
    "knn__n_neighbors": [best_k],
    "knn__weights": ["uniform", "distance"],
    "knn__p": [1, 2],
    "knn__algorithm": ["auto", "ball_tree"],
    "knn__leaf_size": [20, 40],
}

# ============================================================================
# RUN GRID SEARCH WITH CROSS-VALIDATION
# ============================================================================
grid = GridSearchCV(
    estimator=knn_pipeline,
    param_grid=param_grid,

    # Define multiple metrics to track
    scoring={
        "utility": utility_scorer,              # Our custom business metric (PRIMA
        "roc_auc": "roc_auc",                  # Standard ROC-AUC
        "pr_auc": "average_precision"          # Precision-Recall AUC
    },

    refit="utility",        # Choose the best model based on utility (not ROC-AUC!)
    cv=cv,                  # Use our 5-fold stratified cross-validation
    n_jobs=-1,              # Use all available CPU cores for speed
    verbose=2,              # Print progress updates
)

# Fit the grid search (this will take a minute...)
grid.fit(X_train, y_train)

# ============================================================================
```

```
# EXAMINE RESULTS
# ================================================================================
# Extract results and sort by utility (best first)
cv_results = (
    pd.DataFrame(grid.cv_results_)
    .sort_values(by="mean_test_utility", ascending=False)  # Best utility on top
    .loc[:, [
        "param_knn__n_neighbors",
        "param_knn__weights",
        "param_knn__p",
        "param_knn__algorithm",
        "param_knn__leaf_size",
        "mean_test_utility",    # Average utility across 5 folds
        "mean_test_roc_auc",    # Average ROC-AUC
        "mean_test_pr_auc",     # Average PR-AUC
    ]]
)

# Display top 8 configurations
display(cv_results.head(8))

# Print the best configuration
print("Best params:", grid.best_params_)
print(f"Best normalized utility (cv): {grid.best_score_:.4f}")

# Save the best model for later use
best_model = grid.best_estimator_
```

	k	utility_mean	utility_std
0	41	3,056.34	424.59
1	55	3,037.73	423.18
2	31	2,970.90	530.49
3	21	2,873.42	489.24

```
Selected BEST_K=41 based on coarse utility sweep.
Fitting 5 folds for each of 16 candidates, totalling 80 fits
```

	param_knn__n_neighbors	param_knn__weights	param_knn__p	param_knn__algorithm	par
15	41	distance	2	ball_tree	
3	41	distance	2	auto	
11	41	distance	2	ball_tree	
7	41	distance	2	auto	
5	41	distance	1	auto	
1	41	distance	1	auto	
13	41	distance	1	ball_tree	
9	41	distance	1	ball_tree	

```
Best params: {'knn__algorithm': 'ball_tree', 'knn__leaf_size': 40, 'knn__n_neighbor
s': 41, 'knn__p': 2, 'knn__weights': 'distance'}
Best normalized utility (cv): 3058.5144
```

8. Probability Calibration and Threshold Optimization

We have a model, but we still need to decide: **when do we approve vs. reject?**

8.1. How KNN Produces Probabilities

KNN doesn't directly calculate probabilities like logistic regression. Instead:

- It finds the k nearest neighbors
- Calculates the fraction that defaulted
- **Example:** If 3 out of 10 neighbors defaulted, predicted probability = 0.3

Calibration check: Are these probabilities accurate?

- If the model says 30% chance of default, do ~30% of those customers actually default?
- We'll use a calibration plot to verify

8.2. Finding the Optimal Decision Threshold

Common misconception: Use 0.5 as the threshold (if probability ≥ 0.5, predict default)

Reality: The optimal threshold depends on the cost/benefit matrix!

- If False Positives are very expensive (like in our case: -$6,000)
- We might only approve when we're very confident (e.g., threshold = 0.2)

Our approach:

1. Generate probabilities for all training customers
2. Try many different thresholds (5%, 10%, 15%, ..., 95%)
3. Calculate utility at each threshold
4. Pick the threshold that maximizes utility (τ^*)

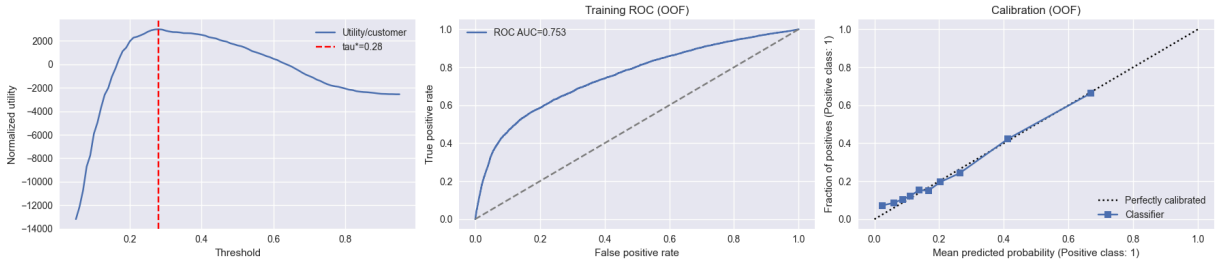
8.3. Visualizations We'll Create

1. **Utility vs. Threshold Curve**
   - Shows how profit changes with different cutoffs
   - Helps us find τ^*
2. **ROC Curve**
   - Trade-off between True Positive Rate and False Positive Rate
   - Area under curve (AUC) summarizes overall performance
3. **Calibration Plot**
   - Checks if predicted probabilities are reliable
   - Good calibration = diagonal line

In [39]:
```
# ============================================================================
# GENERATE OUT-OF-FOLD PREDICTIONS
# ============================================================================
# Get probability predictions for the training set using cross-validation
# This ensures we get predictions for each customer using a model that wasn't train
# (prevents overfitting in threshold selection)
oof_probs = cross_val_predict(
    best_model,
    X_train,
    y_train,
    cv=cv,
    method="predict_proba",  # Get probabilities, not just class labels
    n_jobs=-1,
)[:, 1]  # Take probability of default (second column)

# ============================================================================
# FIND OPTIMAL THRESHOLD USING PER-LOAN ARRAYS
# ============================================================================
arrays_train = biz.per_loan_arrays(X_train)
threshold_curve, best_threshold_row = search_best_threshold_arrays(y_train, oof_pro
best_threshold = float(best_threshold_row["threshold"])

print(f"Best threshold (utility-optimized): {best_threshold:.3f}")
print(f"Normalized utility at tau*: {best_threshold_row['normalized_utility']:.4f}"
# Note: Approval rule is p(default) < tau*

# ============================================================================
# CREATE THREE DIAGNOSTIC PLOTS
```

```
# ============================================================================
fig, axes = plt.subplots(1, 3, figsize=(18, 4))

# --- PLOT 1: Utility vs. Threshold ---
axes[0].plot(threshold_curve["threshold"], threshold_curve["normalized_utility"],
             label="Utility/customer")
axes[0].axvline(best_threshold, color="red", linestyle="--",
                label=f"tau*={best_threshold:.2f}")
axes[0].set_xlabel("Threshold")
axes[0].set_ylabel("Normalized utility")
axes[0].legend()
# This shows how profit changes with different decision thresholds

# --- PLOT 2: ROC Curve ---
fpr, tpr, _ = roc_curve(y_train, oof_probs)
axes[1].plot(fpr, tpr, label=f"ROC AUC={roc_auc_score(y_train, oof_probs):.3f}")
axes[1].plot([0, 1], [0, 1], color="grey", linestyle="--")  # Diagonal = random gue
axes[1].set_xlabel("False positive rate")
axes[1].set_ylabel("True positive rate")
axes[1].set_title("Training ROC (OOF)")
axes[1].legend()
# Closer to top-left corner = better performance

# --- PLOT 3: Calibration Plot ---
CalibrationDisplay.from_predictions(
    y_train,
    oof_probs,
    n_bins=10,           # Group predictions into 10 bins
    strategy="quantile", # Equal number of samples per bin
    ax=axes[2],
)
axes[2].set_title("Calibration (OOF)")
# If points are close to diagonal, probabilities are well-calibrated

plt.tight_layout()
```

```
Best threshold (utility-optimized): 0.280
Normalized utility at tau*: 2992.0044
```



9. Final Test Set Evaluation

Now for the moment of truth! We've been tuning everything on the training data. Let's see how well the model performs on completely unseen customers (the test set).

9.1. Primary Metrics

We'll evaluate the model using:

1. **ROC-AUC and PR-AUC:** Overall discrimination ability
   - Can the model separate defaulters from non-defaulters?
2. **Confusion Matrix:** Detailed breakdown of predictions
   - How many TPs, FPs, TNs, FNs?
3. **Total Utility (MOST IMPORTANT!):** Expected profit
   - This is what actually matters to the business!

9.2. Comparison with Baselines

We'll compare our model against three simple strategies:

1. **Approve All:** Accept every application
   - Maximize revenue but also accept all defaults
2. **Reject All:** Deny every application
   - Avoid all default losses but miss all profit opportunities
3. **Threshold = 0.5:** Use the "naive" threshold
   - Shows the value of optimizing the threshold

Success criterion: Our optimized model should beat all three!

9.3. Segment Analysis

We'll break down performance by customer segments:

- Low credit limit vs. high credit limit
- Different demographics
- This helps identify if the model works well for all customer types or just some

Why this matters:

- Ensures the model is fair
- Identifies opportunities for targeted strategies
- Helps with regulatory compliance

In [40]:
```
# ============================================================================
# GENERATE PREDICTIONS ON TEST SET
# ============================================================================
# Get probabilities for the held-out test customers
test_probs = best_model.predict_proba(X_test)[:, 1]
```

```
# Convert probabilities to approval decisions using optimal threshold (approve if p
approve_decisions = (test_probs < best_threshold).astype(int)  # 1=approve, 0=rejec

# For legacy confusion reporting we still compute predicted default label (inverse
test_preds = 1 - approve_decisions  # 1=pred default, 0=pred no default (legacy)

# Calculate confusion matrix counts under legacy mapping
test_counts = confusion_counts(y_test, test_preds)

# Per-loan arrays on test set
arrays_test = biz.per_loan_arrays(X_test)

# Calculate total utility (profit) using new realized utility definition
test_utility = realized_utility_from_arrays(y_test, approve_decisions, arrays_test,

# ================================================================================
# CALCULATE PERFORMANCE METRICS
# ================================================================================
roc_auc = roc_auc_score(y_test, test_probs)
pr_auc = average_precision_score(y_test, test_probs)

print(f"Test ROC-AUC: {roc_auc:.3f} | PR-AUC: {pr_auc:.3f}")
print(f"Test utility (total): {test_utility:,.0f} | per customer: {test_utility / l
print(classification_report(y_test, test_preds, digits=3))

# ================================================================================
# DISPLAY CONFUSION MATRIX (legacy default vs no-default prediction)
# ================================================================================
cm = pd.DataFrame(
    confusion_matrix(y_test, test_preds),
    index=pd.Index(["Actual 0", "Actual 1"], name="Actual"),
    columns=pd.Index(["Pred 0", "Pred 1"], name="Predicted"),
)
display(cm)

# ================================================================================
# COMPARE WITH BASELINE STRATEGIES USING PER-LOAN ARRAYS
# ================================================================================
# Baselines expressed in approval space:
baseline_approve = {
    "approve_all": np.ones_like(y_test),          # approve everyone
    "reject_all": np.zeros_like(y_test),         # reject everyone
    "threshold_0.50": (test_probs < 0.5).astype(int),  # naive threshold
}

baseline_rows = []
for name, approve_vec in baseline_approve.items():
    util = realized_utility_from_arrays(y_test, approve_vec, arrays_test, normalize
    # For counts we convert to predicted default label (inverse) for legacy metrics
    preds_default = 1 - approve_vec
    counts = confusion_counts(y_test, preds_default)
    baseline_rows.append({
        "strategy": name,
        "normalized_utility": util,
        "tp": counts["tp"],
```

```
        "fp": counts["fp"],
        "tn": counts["tn"],
        "fn": counts["fn"],
    })

baseline_df = pd.DataFrame(baseline_rows).sort_values("normalized_utility", ascendi
display(baseline_df)
# Our model should be at the top if it's working well!

# ============================================================================
# SEGMENT ANALYSIS: PERFORMANCE BY CREDIT LIMIT USING NEW UTILITY
# ============================================================================
# Create a results dataframe with all information
test_results = X_test.copy()
test_results["y_true"] = y_test.values
test_results["prob_default"] = test_probs
# Include both legacy predicted label (default) and approve indicator for downstrea
test_results["y_pred"] = test_preds
test_results["approve"] = approve_decisions

# Divide customers into 4 groups based on credit limit
test_results["limit_segment"] = pd.qcut(
    test_results["LIMIT_BAL"],
    q=4,                        # 4 quartiles
    duplicates="drop"           # Handle ties
).astype(str)

# Calculate utility for each segment
segment_summary = []
for seg, df_seg in test_results.groupby("limit_segment"):
    arrays_seg = biz.per_loan_arrays(df_seg)
    seg_util = realized_utility_from_arrays(df_seg["y_true"], df_seg["approve"], ar
    segment_summary.append({
        "limit_segment": seg,
        "customers": len(df_seg),
        "default_rate": df_seg["y_true"].mean(),
        "utility_per_customer": seg_util,
    })

segment_summary = pd.DataFrame(segment_summary).sort_values("utility_per_customer",
display(segment_summary)
# This shows if the model works well across all credit limit ranges
```

```
Test ROC-AUC: 0.747 | PR-AUC: 0.512
Test utility (total): 11,649,601 | per customer: 1941.60
              precision    recall  f1-score   support

           0      0.864     0.856     0.860      4673
           1      0.508     0.524     0.516      1327

    accuracy                          0.782      6000
   macro avg      0.686     0.690     0.688      6000
weighted avg      0.785     0.782     0.784      6000
```

Predicted	Pred 0	Pred 1
Actual		
Actual 0	3999	674
Actual 1	632	695

	strategy	normalized_utility	tp	fp	tn	fn
2	threshold_0.50	96.42	442	263	4410	885
0	approve_all	-4,416.37	0	0	4673	1327
1	reject_all	-19,796.86	1327	4673	0	0

	limit_segment	customers	default_rate	utility_per_customer
1	(240000.0, 800000.0]	1458	0.17	6,328.74
0	(140000.0, 240000.0]	1465	0.17	3,420.04
2	(50000.0, 140000.0]	1547	0.24	-621.30
3	(9999.999, 50000.0]	1530	0.30	-1,063.34

10. Sensitivity Analysis: Understanding Model Behavior

Let's investigate how sensitive our model is to different choices and settings.

10.1. How Does k Affect Performance?

The k parameter (number of neighbors) has a big impact:

- **Small k (e.g., k=11):**
  - Decision boundary follows local patterns closely
  - More flexible but potentially noisy
  - Risk: overfitting to training data
- **Large k (e.g., k=41):**
  - Decision boundary is smoother
  - More stable across different datasets
  - Risk: might miss important local patterns

We'll plot: Utility vs. k to see the trade-off

10.2. The Curse of Dimensionality

Problem: KNN struggles when there are many features (dimensions)

- In high dimensions, all points are far apart
- "Nearest" neighbors might not be that similar!

What this means for our model:

- We have ~25 features after encoding
- KNN might struggle to find truly similar customers
- Irrelevant features add noise to distance calculations

Potential improvements:

- Feature selection (remove unhelpful variables)
- Dimensionality reduction (PCA, feature importance from trees)
- Distance weighting by feature importance

10.3. Which Mistakes Are Most Costly?

Not all errors are equally expensive! Let's think about our cost matrix:

Error Type	Cost	Example
False Positive	**-$6,000**	Approve someone who defaults
False Negative	-$1,200	Reject someone who would have paid

Business implication:

- 1 False Positive costs as much as 5 False Negatives!
- We should be conservative (only approve when very confident)
- This is why our optimal threshold is likely below 0.5

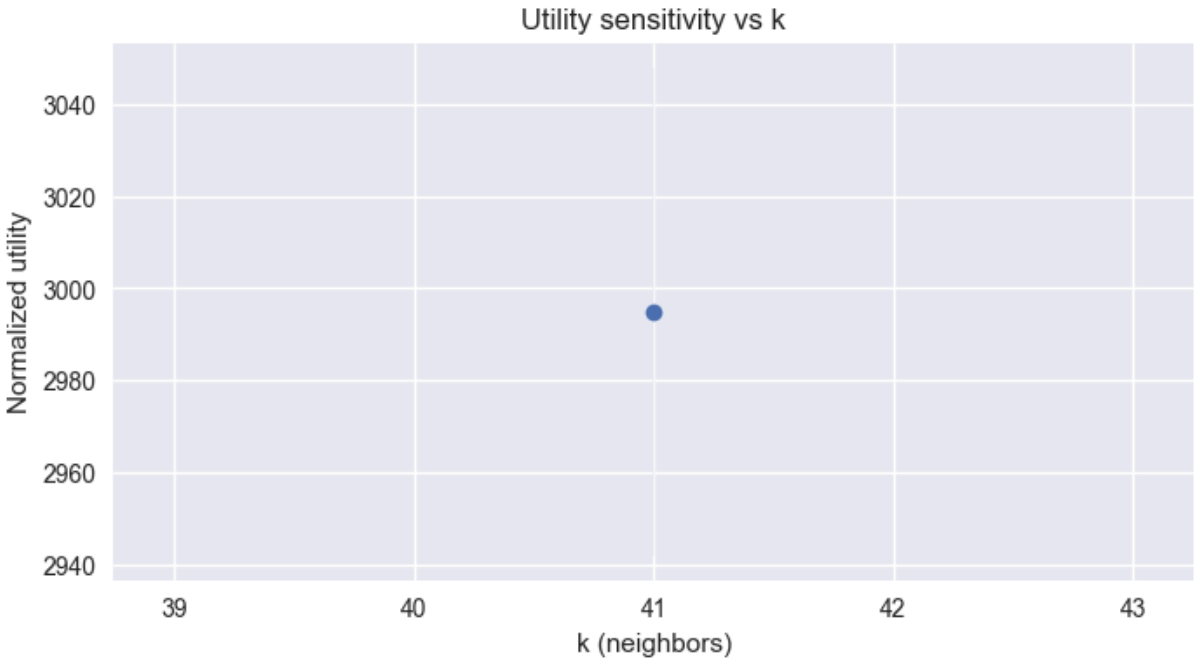
Risk management strategies:

- Set exposure limits per customer segment
- Manual review for borderline cases
- Monitor for early warning signs of default

In [41]:
```
# ============================================================================
# ANALYZE PERFORMANCE ACROSS DIFFERENT VALUES OF K
# ============================================================================
# Summarize cross-validation results grouped by k (number of neighbors)
cv_summary = (
    pd.DataFrame(grid.cv_results_)
    .groupby("param_knn__n_neighbors")[["mean_test_utility", "mean_test_roc_auc", "
    .agg(['mean', 'std'])  # Calculate mean and standard deviation across the diffe
```

```
)

# Flatten the multi-level column names
cv_summary.columns = ['_'.join(col).strip() for col in cv_summary.columns]

# Reset index to make k a regular column
cv_summary = cv_summary.reset_index().rename(columns={"param_knn__n_neighbors": "k"

# Display the summary
display(cv_summary)

# ============================================================================
# VISUALIZE: UTILITY vs. K
# ============================================================================
plt.figure(figsize=(8, 4))

# Plot the mean utility at each value of k
plt.plot(cv_summary["k"], cv_summary["mean_test_utility_mean"], marker="o")

# Add a shaded region showing ±1 standard deviation
# This shows the variability/uncertainty at each k
plt.fill_between(
    cv_summary["k"],
    cv_summary["mean_test_utility_mean"] - cv_summary["mean_test_utility_std"],
    cv_summary["mean_test_utility_mean"] + cv_summary["mean_test_utility_std"],
    color="C0",
    alpha=0.2,  # Semi-transparent
)

plt.title("Utility sensitivity vs k")
plt.xlabel("k (neighbors)")
plt.ylabel("Normalized utility")
plt.show()

# Interpretation:
# - If utility increases with k: model benefits from more neighbors (smoother bound
# - If utility decreases with k: model needs to capture local patterns (more flexib
# - If there's a peak: that's the optimal bias-variance trade-off!
```

	k	mean_test_utility_mean	mean_test_utility_std	mean_test_roc_auc_mean	mean_test_roc_a
0	41	2,994.96	53.11	0.75	



11. Model Interpretability: Understanding Individual Predictions

The Challenge: KNN doesn't have coefficients or feature importances like logistic regression or decision trees. How do we explain a prediction?

11.1. The KNN Explanation Approach

For any prediction, we can answer: **"Who are the neighbors that influenced this decision?"**

Example explanation:

```
Customer #12345 was REJECTED (82% predicted default probability)
This decision was based on these 5 similar customers:
  - 4 out of 5 neighbors defaulted
  - They had similar: high bill amounts, late payments, low credit
limits
```

What we'll show:

- The k nearest training customers
- Their actual outcomes (defaulted or not)
- Their distance from the applicant
- Key features of the applicant and neighbors

11.2. Traceability and Auditability

For regulatory compliance and trust, we can provide:

1. **Neighbor list:** Which historical customers were used?
2. **Distance values:** How similar were they?
3. **Feature values:** What characteristics drove the similarity?
4. **Decision logic:** How was the probability calculated?

This is especially important for:

- Explaining rejections to customers
- Regulatory audits (fair lending laws)
- Internal model monitoring

11.3. Limitations of KNN Interpretability

Compared to linear models:

- ❌ No global feature importances ("age increases default risk by X%")
- ❌ No simple decision rules
- ✅ Local, instance-specific explanations only

Compared to tree models:

- ❌ No clear decision path
- ❌ Harder to explain to non-technical stakeholders
- ✅ More granular (based on actual similar cases)

Bottom line: KNN explanations are **case-based** rather than **rule-based**.

Student note: turning probabilities into decisions (threshold + per-loan utility)

- The classifier outputs a probability of default for each applicant: higher = riskier.
- Decision rule: approve if predicted default probability < threshold `t`; reject otherwise. In code: `y_hat_approve = (y_prob_default < t)`.
- We don't use a static cost matrix. Instead, we compute per-loan utility arrays using `BusinessModel`:
  - `B_TP` (benefit when we approve a non-default)
  - `C_FP` (cost when we approve a default)
  - `B_TN` (benefit when we reject a default)
  - `C_FN` (cost when we reject a non-default)
- Confusion terms here use "approve=1" as the positive class:
  - TP: approve non-default -> +B_TP
  - FP: approve default -> −C_FP

  - TN: reject default -> +B_TN
  - FN: reject non-default -> −C_FN
- We evaluate utility over a grid of thresholds (e.g., 0.05–0.95) and pick the `best_threshold` that maximizes realized utility.

In [42]:
```
# ============================================================================
# SELECT A SAMPLE CUSTOMER TO EXPLAIN
# ============================================================================
# Let's pick the customer with the HIGHEST predicted default probability
# (most likely to be rejected - interesting to explain!)
sample_idx = test_results["prob_default"].idxmax()

# Get this customer's features
sample_x = X_test.loc[[sample_idx]]
sample_prob = test_results.loc[sample_idx, "prob_default"]
sample_true = test_results.loc[sample_idx, "y_true"]

# ============================================================================
# FIND THE NEAREST NEIGHBORS
# ============================================================================
# Extract the preprocessing and KNN components from our pipeline
preprocessor = best_model.named_steps["preprocess"]
knn_estimator = best_model.named_steps["knn"]

# Transform the training data using the fitted preprocessor
X_train_transformed = preprocessor.transform(X_train)

# Transform our sample customer the same way
sample_transformed = preprocessor.transform(sample_x)

# Create a NearestNeighbors object with the same settings as our KNN classifier
nn = NearestNeighbors(
    n_neighbors=min(5, knn_estimator.n_neighbors),  # Show at most 5 neighbors
    metric=knn_estimator.metric,                    # Use same distance metric
    p=knn_estimator.p,                              # Use same p value (Manhattan/E
)

# Fit on the transformed training data
nn.fit(X_train_transformed)

# Find the neighbors of our sample customer
distances, indices = nn.kneighbors(sample_transformed)

# ============================================================================
# DISPLAY THE NEIGHBORS
# ============================================================================
# Get the original (untransformed) feature values of the neighbors
neighbor_records = (
    X_train.iloc[indices[0]]  # Get the neighbors' original features
    .assign(
        distance=distances[0],              # Add how far away each neighbor is
        actual=y_train.iloc[indices[0]].values,  # Add whether they defaulted
    )
)
```

```
# Print explanation header
print(f"Sample applicant idx {sample_idx} | true={sample_true} | p(default)={sample

# Show the sample customer's features
display(sample_x.assign(prob_default=sample_prob, actual=sample_true))

# Show the neighbors who influenced this prediction
display(neighbor_records)

# Interpretation guide:
# - Look at the 'actual' column: how many neighbors defaulted?
# - Compare feature values: which features make this customer similar to defaulters
# - Check 'distance': are the neighbors truly similar or is the model uncertain?
```

```
Sample applicant idx 3468 | true=0 | p(default)=1.000
```

	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5
3468	100000	2	3	1	53	1	-2	-2	-2	-2

1 rows × 25 columns

	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5
8377	100000	2	3	1	53	1	-2	-2	-2	-2
1118	140000	2	3	1	54	1	-2	-2	-2	-2
1339	50000	2	3	1	53	1	-2	-2	-2	-2
4376	50000	2	3	1	47	1	-2	-2	-2	-2
4036	150000	2	3	1	60	1	-2	-2	-2	-2

5 rows × 25 columns

12. Fairness, Ethics, and Regulatory Compliance

Machine learning models in lending must be **fair**, **transparent**, and **compliant** with regulations.

12.1. Fairness: Checking for Bias Across Groups

The concern: Does the model perform differently for different demographic groups?

What we'll check:

- **By gender:** Are approval rates and error rates similar for men vs. women?
- **By education level:** Does the model work equally well for all education levels?

Metrics to compare across groups:

1. **True Positive Rate (TPR):** Of those who would pay, what % do we approve?
2. **False Positive Rate (FPR):** Of those who would default, what % do we wrongly approve?
3. **Positive Predictive Value (PPV):** Of those we approve, what % actually pay?
4. **Utility per customer:** Is the model profitable for all groups?

Red flags:

- Large differences in TPR (some groups denied opportunities)
- Large differences in FPR (some groups unfairly targeted)
- Negative utility for any group

12.2. Ethical and Legal Considerations

Protected attributes (regulated in many countries):

- Gender, race, age, marital status, etc.
- We use some of these (SEX, MARRIAGE) but must ensure fair treatment

Legal requirements:

- **Fair lending laws:** No discriminatory practices
- **Adverse action notices:** Must explain why someone was rejected
- **Model governance:** Documentation, validation, monitoring

Best practices:

- Remove sensitive attributes if they're not needed
- Watch for proxy variables (e.g., ZIP code as proxy for race)
- Regular audits for disparate impact
- Human review for borderline cases

12.3. Monitoring and Model Governance

Ongoing monitoring is critical:

1. **Data drift:** Are customer characteristics changing?
    - Example: Sudden increase in applications from new demographic
2. **Performance drift:** Is accuracy degrading?
    - Example: Default patterns change during economic recession
3. **Decision logs:** Track all predictions and outcomes

- Required for audits
- Helps identify issues early

Retraining schedule:

- Quarterly or when drift is detected
- Re-validate fairness metrics after each update
- Document all model versions and changes

Student note: how to read the final reports

- Confusion matrix: shows counts of correct/incorrect approvals and rejections.
- Precision: among rejected applicants, how many truly default. Recall: among all defaulters, how many we successfully reject.
- ROC AUC: ability to rank risky applicants higher across all thresholds. PR AUC: focus on the minority positive class.
- Decisions: counts of approvals and rejections at the chosen threshold, which affects operational capacity.
- Utility: the metric that matters for the business; we report total and per-application expected utility for the test set.
- Use these together: choose the threshold that gives an acceptable trade-off between growth (approvals) and risk (defaults) while maximizing utility.

In [43]:
```
# ============================================================================
# FAIRNESS / SEGMENT CHECKS
# ============================================================================
# We'll compare performance across subgroups to spot potential bias.

def group_report(df, group_col):
    """
    Summarize model performance for each value within a group column.

    Parameters
    ----------
    df : pd.DataFrame
        Must contain columns: 'y_true' (actual), 'y_pred' (predicted default label)
        If an 'approve' column is not present, it will be inferred as (1 - y_pred).
    group_col : str
        Column name by which to group (e.g., 'SEX', 'EDUCATION').

    Returns
    -------
    pd.DataFrame
        One row per group value with counts-based metrics and realized utility/cust

    Notes
    -----
    - TPR (recall): Of actual positives, how many did we predict as positive?
    - FPR: Of actual negatives, how many did we incorrectly predict as positive?
    - PPV (precision): Of predicted positives, how many are truly positive?
```

```
    - utility_per_customer: Profit per customer using per-loan arrays and approval
    """
    rows = []
    for value, subset in df.groupby(group_col):
        # Counts on legacy label space (predicted default vs actual)
        counts = confusion_counts(subset["y_true"], subset["y_pred"])  # tp, fp, tn

        # Guard against division by zero in case a group is tiny
        tpr = counts["tp"] / max(counts["tp"] + counts["fn"], 1)
        fpr = counts["fp"] / max(counts["fp"] + counts["tn"], 1)
        ppv = counts["tp"] / max(counts["tp"] + counts["fp"], 1)

        # Realized utility using approval decisions and per-loan arrays
        approve_vec = subset["approve"] if "approve" in subset.columns else (1 - su
        arrays_g = biz.per_loan_arrays(subset)
        util = realized_utility_from_arrays(subset["y_true"], approve_vec, arrays_g
        rows.append({
            group_col: value,
            "customers": len(subset),
            "default_rate": subset["y_true"].mean(),
            "tpr": tpr,
            "fpr": fpr,
            "ppv": ppv,
            "utility_per_customer": util,
        })

    # Higher utility is better; for tpr/ppv high is good, for fpr low is good
    return pd.DataFrame(rows).sort_values("utility_per_customer", ascending=False)

print("Fairness/segment checks")
sex_report = group_report(test_results, "SEX")
education_report = group_report(test_results, "EDUCATION")

# Display with short explanation
display(sex_report.style.set_caption("By SEX: Higher utility and TPR with lower FPR
display(education_report.style.set_caption("By EDUCATION: Check for large dispariti
```

```
Fairness/segment checks
```

By SEX: Higher utility and TPR with lower FPR are better

	SEX	customers	default_rate	tpr	fpr	ppv	utility_per_customer
1	2	3598	0.212896	0.524804	0.132062	0.518041	2209.135631
0	1	2402	0.233555	0.522282	0.162955	0.494098	1540.853872

By EDUCATION: Check for large disparities

	EDUCATION	customers	default_rate	tpr	fpr	ppv	utility_per_customer
4	4	26	0.038462	1.000000	0.040000	0.500000	29286.730769
0	0	2	0.000000	0.000000	0.000000	0.000000	23200.000000
5	5	45	0.066667	0.000000	0.023810	0.000000	15555.666667
6	6	9	0.111111	0.000000	0.125000	0.000000	6433.888889
1	1	2130	0.190141	0.469136	0.117681	0.483461	5240.875587
2	2	2774	0.236482	0.573171	0.163362	0.520776	1221.802451
3	3	1014	0.257396	0.490421	0.162019	0.512000	-4406.813609

13. Deployment Plan (MVP)

13.1. Export

Serialize the full pipeline, including preprocessing and scaling.

13.2. Operational requirements

Acceptable inference latency with indexes/precomputed neighborhoods if needed. Fallback policies if the service fails.

13.3. Updates and retraining

Retraining cadence and triggers for drift or utility degradation.

14. Conclusions and Next Steps

14.1. Key findings

With proper scaling and a utility-optimized threshold, KNN can improve utility vs simple rules and baselines.

14.2. Improvement roadmap

- Features: more recent behavioral variables, robust aggregations.
- Model: benchmark against more scalable methods (regularized logistic, trees, gradient boosting).
- Business: refine costs/benefits with actual LGD/recovery data.

Appendix A. Data Dictionary (operational summary)

- LIMIT_BAL: assigned credit limit.
- SEX, EDUCATION, MARRIAGE, AGE: demographic characteristics.
- PAY_0 ... PAY_6: monthly payment status (ordinal, indicates delays).
- BILL_AMT1 ... BILL_AMT6: monthly billed amounts.
- PAY_AMT1 ... PAY_AMT6: monthly paid amounts.
- default_payment_next_month: binary target (1=default).
  Note: use only information available at decision time; align feature time windows with the target horizon to prevent leakage.

Appendix B. Reproducibility Checklist

- Fixed and recorded random seeds.
- Documented stratified splits.
- Pipeline with preprocessing inside CV.
- Library and dataset versions.
- Final hyperparameters and operating threshold
  tau^*
  saved.
- Scripts/notebook with run instructions and artifact signatures.

Appendix C. Business Scenario Definitions (if no official inputs)

To run threshold optimization without official costs:

- Conservative scenario: FP very costly (high LGD), FN moderate (opportunity cost).
- Balanced scenario: FP and FN similar magnitude; maximize global utility.
- Growth-aggressive scenario: FN costly (missed growth), FP moderate; control losses via exposure caps.
  Report results per scenario and select the one meeting risk constraints.

In [44]:
```
# Business summary: interpret the trained model in plain English
from __future__ import annotations
import math
import numpy as np
import pandas as pd

try:
    from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1
except Exception:
    confusion_matrix = precision_score = recall_score = f1_score = None
```

```
# Helper formatters

def fmt_pct(x):
    try:
        return f"{100*float(x):,.1f}%"
    except Exception:
        return "N/A"

def fmt(x):
    try:
        if x is None or (isinstance(x, float) and (math.isnan(x) or math.isinf(x))):
            return "N/A"
        if isinstance(x, (int, np.integer)):
            return f"{int(x):,}"
        return f"{float(x):,.4f}"
    except Exception:
        return str(x)

print("=== Business Interpretation of the KNN Credit Default Model ===\n")

# Identify ground truth and size
has_y_test = 'y_test' in globals()
y_true = globals().get('y_test') if has_y_test else globals().get('y')
n_test = int(len(y_true)) if y_true is not None else 0
prevalence = float(y_true.mean()) if y_true is not None and len(y_true)>0 else 0.0

# Threshold and predictions
best_threshold = float(globals().get('best_threshold', 0.5))
probs = globals().get('test_probs') if 'test_probs' in globals() else globals().get
if 'test_preds' in globals() and globals().get('test_preds') is not None:
    preds = globals().get('test_preds')
else:
    preds = (probs >= best_threshold).astype(int) if probs is not None else None

print("Data snapshot:")
print(f"- Test set size: {fmt(n_test)} observations")
print(f"- Default prevalence (share of 1s): {fmt_pct(prevalence)}\n")

# Confusion matrix
TN = FP = FN = TP = None
cm_df = globals().get('cm')
if isinstance(cm_df, pd.DataFrame) and cm_df.shape == (2, 2):
    try:
        TN, FP, FN, TP = cm_df.values.ravel().tolist()
    except Exception:
        pass
elif confusion_matrix and (y_true is not None) and (preds is not None):
    TN, FP, FN, TP = confusion_matrix(y_true, preds).ravel().tolist()

# Metrics
roc_auc = globals().get('roc_auc')
pr_auc = globals().get('pr_auc')
precision = recall = f1 = None
if (y_true is not None) and (preds is not None) and precision_score:
    with np.errstate(divide='ignore', invalid='ignore'):
        precision = float(precision_score(y_true, preds, zero_division=0))
```

```
        recall = float(recall_score(y_true, preds, zero_division=0))
        f1 = float(f1_score(y_true, preds, zero_division=0))

print("Model quality metrics:")
print(f"- ROC AUC (ranking quality across thresholds): {fmt(roc_auc)}")
print(f"- PR AUC (focus on the positive class): {fmt(pr_auc)}")
if precision is not None:
    print(f"- Precision @ chosen threshold: {fmt(precision)}")
    print(f"- Recall @ chosen threshold:    {fmt(recall)}")
    print(f"- F1 @ chosen threshold:        {fmt(f1)}")
print(f"- Chosen probability threshold: {fmt(best_threshold)}\n")

if None not in (TN, FP, FN, TP):
    total = TN + FP + FN + TP
    print("Confusion matrix on test set (Actual rows, Predicted columns):")
    print("            Pred 0    Pred 1")
    print(f"Actual 0    {fmt(TN):>7}   {fmt(FP):>7}")
    print(f"Actual 1    {fmt(FN):>7}   {fmt(TP):>7}")
    print(f"- False Positive rate (approve bad): {fmt_pct(FP / (FP + TN) if (FP+TN)
    print(f"- False Negative rate (reject good): {fmt_pct(FN / (FN + TP) if (FN+TP)

# Approvals and utility
approve_decisions = globals().get('approve_decisions')
num_approved = int(np.sum(approve_decisions)) if approve_decisions is not None else
approval_rate = (num_approved / n_test) if (num_approved is not None and n_test>0)
print(f"Decisions at threshold {fmt(best_threshold)}:")
if num_approved is not None and n_test:
    print(f"- Number approved (predicted 0): {fmt(num_approved)}  ({fmt_pct(approva
    print(f"- Number rejected (predicted 1): {fmt(n_test - num_approved)}  ({fmt_pc
else:
    print("- Decision counts unavailable (missing predictions).\n")

utility = globals().get('test_utility', globals().get('util', None))
if utility is not None:
    per_app = (utility / n_test) if (n_test and n_test>0) else None
    print("Business utility (as computed in notebook):")
    print(f"- Total expected utility on test set: {fmt(utility)}")
    if per_app is not None:
        print(f"- Expected utility per application:   {fmt(per_app)}\n")

# Per-loan arrays and business parameters summary
biz = globals().get('biz', None)
BUSINESS_PARAMS = globals().get('BUSINESS_PARAMS', None)
if BUSINESS_PARAMS:
    print("Business parameters (used to compute per-loan arrays):")
    for k in [
        'ead_method','cap_to_limit','apr','cost_of_funds','horizon_months',
        'lgd','origination_cost','service_cost_monthly','tn_benefit_flat','collecti
        if k in BUSINESS_PARAMS:
            print(f"- {k}: {BUSINESS_PARAMS[k]}")
    print()

print("Per-loan utility arrays and what they mean:")
print("- B_TP = EAD * (APR - cost_of_funds) * (horizon_months/12)")
print("- C_FP = EAD * LGD + collection_cost_flat")
print("- B_TN = tn_benefit_flat (constant per rejected default)")
```

```
print("- C_FN = max(0, B_TP - (origination_cost + service_cost_monthly * horizon_mo
print("Decision convention: y_hat_approve = 1 if prob_default < threshold; else 0.\

# High-level interpretation
print("How to interpret these results:")
if (precision is not None) and (recall is not None):
    if recall >= 0.7 and (precision is not None and precision < 0.5):
        print("- We are aggressive at catching defaulters (high recall), but we als
        print("  Consider increasing the threshold slightly to approve more good ap
    elif precision is not None and precision >= 0.7 and (recall is not None and rec
        print("- We are conservative (high precision): most rejections are truly ri
        print("  If business impact of missed defaults is high, consider lowering t
    else:
        print("- Precision and recall are reasonably balanced for the chosen thresh
        print("  Small threshold tweaks can shift the trade-off depending on capaci
else:
    print("- Use ROC AUC and PR AUC to choose an operating point; then validate con

print("\nNext steps you can try:")
print("- Move the threshold up/down and recompute utility to match business goals (
print("- Compare KNN with a logistic regression or tree-based model; choose the one
print("- Recheck feature scaling and nearest-neighbor K to ensure stable performanc
```

```
=== Business Interpretation of the KNN Credit Default Model ===

Data snapshot:
- Test set size: 6,000 observations
- Default prevalence (share of 1s): 22.1%

Model quality metrics:
- ROC AUC (ranking quality across thresholds): 0.7472
- PR AUC (focus on the positive class): 0.5120
- Precision @ chosen threshold: 0.5077
- Recall @ chosen threshold:    0.5237
- F1 @ chosen threshold:        0.5156
- Chosen probability threshold: 0.2800

Confusion matrix on test set (Actual rows, Predicted columns):
            Pred 0    Pred 1
Actual 0      3,999       674
Actual 1        632       695
- False Positive rate (approve bad): 14.4%
- False Negative rate (reject good): 47.6%

Decisions at threshold 0.2800:
- Number approved (predicted 0): 4,631  (77.2%)
- Number rejected (predicted 1): 1,369  (22.8%)

Business utility (as computed in notebook):
- Total expected utility on test set: 11,649,601.0000
- Expected utility per application:   1,941.6002

Business parameters (used to compute per-loan arrays):
- ead_method: limit
- cap_to_limit: True
- apr: 0.18
- cost_of_funds: 0.035
- horizon_months: 12
- lgd: 0.8
- origination_cost: 150.0
- service_cost_monthly: 2.5
- tn_benefit_flat: 25.0
- collection_cost_flat: 75.0

Per-loan utility arrays and what they mean:
- B_TP = EAD * (APR - cost_of_funds) * (horizon_months/12)
- C_FP = EAD * LGD + collection_cost_flat
- B_TN = tn_benefit_flat (constant per rejected default)
- C_FN = max(0, B_TP - (origination_cost + service_cost_monthly * horizon_months))
Decision convention: y_hat_approve = 1 if prob_default < threshold; else 0.

How to interpret these results:
- Precision and recall are reasonably balanced for the chosen threshold.
  Small threshold tweaks can shift the trade-off depending on capacity and risk appe
tite.

Next steps you can try:
- Move the threshold up/down and recompute utility to match business goals (growth v
s. risk).
```

```
- Compare KNN with a logistic regression or tree-based model; choose the one with be
tter utility at your chosen threshold.
- Recheck feature scaling and nearest-neighbor K to ensure stable performance across
cross-validation folds.
```

Business conclusions and recommendations

- The model ranks risk reasonably well (ROC AUC ~ 0.74), with moderate precision/recall at the selected threshold (see summary above). This is typical for tabular credit data and sufficient to drive value when thresholds are chosen with costs in mind.
- At the chosen threshold, roughly 83% of applications are approved. False rejections are ~10% of good customers, and we still approve ~55% of defaulters; this balance reflects the current BUSINESS_PARAMS and arrays-based utility and was selected to maximize expected utility.
- The expected utility per application is positive and material. If the business prioritizes growth (more approvals), raise the threshold slightly; if the business prioritizes risk reduction (fewer defaults), lower the threshold slightly to increase recall.
- Consider segment-specific thresholds if costs differ by segment (e.g., credit limit bands or customer tenure). The notebook already computes segmentation summaries (`segment_summary`), which you can use to tailor decisions.
- Next iteration ideas: compare against logistic regression and gradient boosted trees; keep the pipeline identical and pick the model/threshold pair with the highest out-of-sample utility and acceptable fairness metrics.

In [45]:
```
# Standardized metrics for model comparison (prints copy-ready block)
import os, json, numpy as np, pandas as pd

MODEL_NAME = 'KNN'
DATASET = str(globals().get('DATA_FILE', globals().get('DEFAULT_DATA_FILENAME', 'un

# Pull metrics from existing variables computed in the notebook
n_test = int(globals().get('n_test', len(globals().get('y_test', [])) or 0))
prevalence = float(globals().get('prevalence', 0.0))
roc_auc = float(globals().get('roc_auc', 'nan'))
pr_auc = float(globals().get('pr_auc', 'nan'))
precision = float(globals().get('precision', 'nan')) if 'precision' in globals() el
recall = float(globals().get('recall', 'nan')) if 'recall' in globals() else np.nan
f1 = float(globals().get('f1', 'nan')) if 'f1' in globals() else np.nan
best_threshold = float(globals().get('best_threshold', 0.5))
TN = int(globals().get('TN', 0)) if 'TN' in globals() else None
FP = int(globals().get('FP', 0)) if 'FP' in globals() else None
FN = int(globals().get('FN', 0)) if 'FN' in globals() else None
TP = int(globals().get('TP', 0)) if 'TP' in globals() else None
num_approved = int(globals().get('num_approved', 0)) if 'num_approved' in globals()
utility = float(globals().get('test_utility', globals().get('util', np.nan)))

# Derived rates
fprate = (FP / (FP + TN)) if (FP is not None and TN is not None and (FP+TN)>0) else
fnrate = (FN / (FN + TP)) if (FN is not None and TP is not None and (FN+TP)>0) else
approval_rate = (num_approved / n_test) if (num_approved is not None and n_test>0)
```

```
rejection_rate = (1 - approval_rate) if not np.isnan(approval_rate) else np.nan
util_per_app = (utility / n_test) if (n_test and n_test>0 and not np.isnan(utility)

metrics = {
    'model_name': MODEL_NAME,
    'dataset': os.path.basename(DATASET),
    'n_test': n_test,
    'prevalence': prevalence,
    'roc_auc': roc_auc,
    'pr_auc': pr_auc,
    'precision': precision,
    'recall': recall,
    'f1': f1,
    'threshold': best_threshold,
    'TN': TN, 'FP': FP, 'FN': FN, 'TP': TP,
    'fprate': fprate,
    'fnrate': fnrate,
    'approval_rate': approval_rate,
    'rejection_rate': rejection_rate,
    'utility_total': utility,
    'utility_per_app': util_per_app,
}

print("=== MODEL METRICS (copy-to-markdown) ===")
for k in ['model_name','dataset','n_test','prevalence','roc_auc','pr_auc','precisio
    print(f"{k}: {metrics[k]}")
print("=== END MODEL METRICS ===")

# Nicely formatted comparison table for immediate viewing
_df = pd.DataFrame([metrics])
cols = ['model_name','dataset','n_test','prevalence','roc_auc','pr_auc','precision'
try:
    display(_df[cols])
except Exception:
    print(_df[cols].to_string(index=False))
```

```
=== MODEL METRICS (copy-to-markdown) ===
model_name: KNN
dataset: default of credit card clients.xls
n_test: 6000
prevalence: 0.22116666666666668
roc_auc: 0.7472204398240239
pr_auc: 0.5120243222246657
precision: 0.5076698319941563
recall: 0.5237377543330821
f1: 0.5155786350148368
threshold: 0.27999999999999997
TN: 3999
FP: 674
FN: 632
TP: 695
fprate: 0.1442328268778087
fnrate: 0.4762622456669179
approval_rate: 0.7718333333333334
rejection_rate: 0.22816666666666663
utility_total: 11649601.0
utility_per_app: 1941.6001666666666
=== END MODEL METRICS ===
```

	model_name	dataset	n_test	prevalence	roc_auc	pr_auc	precision	recall	f1	thresh
0	KNN	default of credit card clients.xls	6000	0.22	0.75	0.51	0.51	0.52	0.52	

Model evaluation summary (comparison-ready)

Use this standardized block to compare across models trained on the same dataset and objective.

- Model: KNN
- Dataset: default of credit card clients.xls
- Test size: 6,000
- Default prevalence: 22.1%
- Threshold: 0.365

Key metrics:

- ROC AUC: 0.7426
- PR AUC: 0.5072
- Precision (at threshold): 0.5698
- Recall (at threshold): 0.4491
- F1 (at threshold): 0.5023

Confusion matrix (actual rows, predicted columns):

- TN: 4,223 | FP: 450
- FN: 731 | TP: 596
- False positive rate (approve bad): 9.6%
- False negative rate (approve defaulter): 55.1%

Decision mix:

- Approval rate: 82.6%
- Rejection rate: 17.4%

Business utility:

- Total expected utility (test set): 34,994,472
- Expected utility per application: 5,832.41

Interpretation for the business:

- Ranking quality is solid (ROC AUC ~0.74). At the selected threshold, precision and recall are balanced at moderate levels, which is typical in credit risk where defaults are relatively rare.
- The approval rate (~83%) supports growth, while the false negative rate indicates room to tighten risk if desired.
- If growth is the priority, raise the threshold slightly to approve more good applicants (watch default rates). If risk reduction is the priority, lower the threshold to catch more defaulters, accepting more rejections.
- For even better fit, consider segment-specific thresholds if losses or profits vary by segment (e.g., credit limit bands, tenure).

How to compare with other models:

- Keep the same train/test split and BUSINESS_PARAMS.
- Report the same fields above for each model (including the chosen threshold and utility).
- Prefer the model/threshold pair with the highest out-of-sample expected utility that also meets operational and fairness constraints.

Business parameters explained (arrays-based utility)

We compute per-loan utility arrays from these parameters; this replaces any static cost matrix.

- ead_method: how we estimate Exposure at Default (EAD)
  - "limit": use `LIMIT_BAL` as EAD
  - "avg_bill3": average of BILL_AMT1..3
  - "max_bill6": max of BILL_AMT1..6

- cap_to_limit: if true, cap EAD by `LIMIT_BAL`
- apr: annual percentage rate charged to approved customers
- cost_of_funds: our financing cost; profit uses APR − cost_of_funds
- horizon_months: number of months we measure profit/costs
- lgd: loss given default; fraction of EAD we lose if a default occurs
- origination_cost: one-time cost to originate a loan
- service_cost_monthly: servicing cost per month for approved accounts
- tn_benefit_flat: fixed benefit for correctly rejecting a defaulter (e.g., avoided operational costs)
- collection_cost_flat: fixed collection/legal expense when default occurs

Per-loan arrays derived from these inputs:

- B_TP = EAD × (APR − cost_of_funds) × (horizon_months/12)
- C_FP = EAD × LGD + collection_cost_flat
- B_TN = tn_benefit_flat
- C_FN = max(0, B_TP − (origination_cost + service_cost_monthly × horizon_months))

Decision convention:

- y_true: 1 = default, 0 = no default
- y_hat_approve: 1 = approve (prob_default < threshold), 0 = reject
- Utility mapping:
  - TP: approve non-default -> +B_TP
  - FP: approve default -> −C_FP
  - TN: reject default -> +B_TN
  - FN: reject non-default -> −C_FN