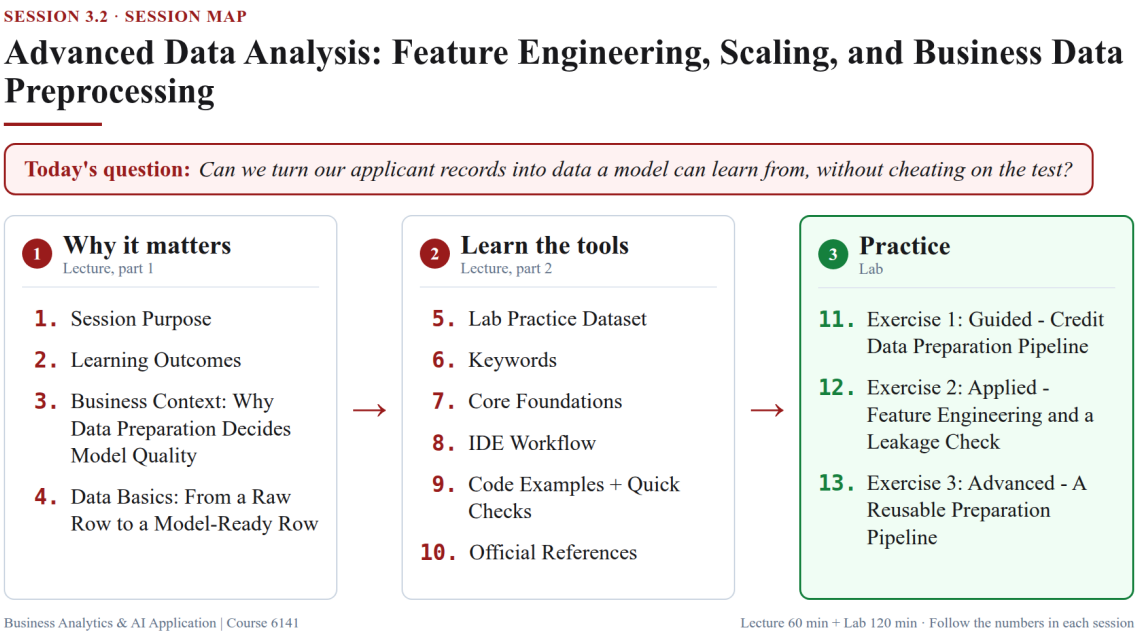
Session 3.2: Advanced Data Analysis: Feature Engineering, Scaling, and Business Data Preprocessing

Course 6141: Business Analytics & AI Application | Reading for Week 10



Schedule: Week 10 | In-person classes (1:2 Delivery Ratio: 60 Min Lecture + 120 Min Lab)

1. Session Purpose

- A bank's credit team wants a model that predicts which loan applicants will default (fail to repay their loan)
- Its applicant records mix text, gaps, and numbers on very different scales, such as an income of `65,000` next to a utilization of `24.5`
- The credit risk manager asks: can we turn our applicant records into data a model can learn from, without cheating on the test? Today, you prepare the data step by step, so that Session 3.3 can train a model on it

2. Expected Learning Outcomes (By the end of this session, you will be able to)

1. "What do we predict, and from what?": split a table into features `X` (the inputs) and the target `y` (the answer to predict)

2. "How do text and gaps become numbers?": fill missing values (imputation), turn text categories into 0/1 columns (one-hot encoding), and put numbers on a common scale (standardization)
3. "How do we test honestly?": split the data into a training set and a test set before learning anything from it, so that no test information leaks into training (data leakage)

3. Business Context: Why Data Preparation Decides Model Quality

- A model is only as good as the data it learns from ("garbage in, garbage out"). Data preparation usually takes more of a project's time than building the model itself.
- A model reads only numbers, with no gaps. Our raw data breaks both rules:
  - `Education_Tier` is text (`Bachelor`, `Master`, `Doctorate`)
  - One income and one utilization value are missing
- A model must be tested on applicants it has never seen, just as a student is tested on new exam questions, not on the practice questions. Otherwise the bank believes the model is better than it is, and approves risky loans.
- Credit utilization: the share of an applicant's credit limit that is in use. `48.2`% means $48.20 of every $100 of available credit is borrowed.

4. Data Basics: From a Raw Row to a Model-Ready Row

- Applicant `CLI-701`, before and after preparation:

Column	Raw value	Model-ready value	Step
`Age_Years_Count`	`45`	`0.41`	standardized
`Annual_Income_USD`	`125000.0`	`0.44`	standardized
`Credit_Utilization_Pct`	`48.2`	`0.82`	standardized
`Education_Tier`	`Master`	`Education_Tier_Master` = `1`, `Education_Tier_Doctorate` = `0`	one-hot encoded

- The standardized values use the training rows' averages and spreads (Micro-Example 3). Micro-Example 3 also standardizes the 0/1 columns; Exercise 3 shows how to scale only the number columns.
- **One-hot encoding:** one 0/1 column per category. With `drop_first=True`, the first category (`Bachelor`) gets no column: a row with `0` in both remaining columns is a Bachelor. Never code categories as `1`, `2`, `3`; that would tell the model a Doctorate is "three times" a Bachelor.
- **Standardization:** `(value - average) / standard deviation`. The result says how many standard deviations a value is above (+) or below (-) the average, so income and utilization become comparable. Standard deviation measures how spread out a column is.
- **Train/test split:** the model learns from the training set; the test set is kept aside and used once, at the end, to measure the model honestly.

5. Lab Practice Dataset

- File Link: `data/session_03_02_dataset.csv`
- Audited 12-row loan applicant dataset (fictional teaching data), one row per applicant, with 2 missing values on purpose:
  - `Client_ID`: Applicant code (e.g., `CLI-701`). An ID, not a feature: it says nothing about the applicant.
  - `Age_Years_Count`: Age in years.
  - `Annual_Income_USD`: Yearly income in U.S. Dollars (missing for `CLI-707`).
  - `Credit_Utilization_Pct`: Share of the credit limit in use, in percent (missing for `CLI-709`).
  - `Education_Tier`: Highest education: `Bachelor`, `Master`, or `Doctorate`.
  - `Defaulted_Binary`: The target: `1` = defaulted, `0` = repaid. 3 of the 12 applicants defaulted.

6. Keywords (Core Programming Concepts)

- `Features (X)` and `Target (y)`:
  - What: `X` holds the input columns; `y` holds the outcome to predict
  - Why: every Scikit-Learn model expects this split
- `Imputation`:
  - What: filling missing values, for example with the column's median (the middle value)
  - Why: models cannot learn from rows with gaps
- `One-Hot Encoding`:
  - What: one 0/1 column per text category
  - Why: turns text into numbers without inventing an order
- `Standardization`:
  - What: rescaling a column to an average of 0 and a standard deviation of 1
  - Why: stops large-unit columns, such as income, from drowning out small ones
- `Train/Test Split`:
  - What: keeping some rows aside to test the model at the end
  - Why: the only honest measure of how the model will do on new applicants
- `Data Leakage`:
  - What: information from the test set slipping into training
  - Why: makes a model look better than it really is

7. Core Programming & Computational Foundations

- **7.1 Features and target: what goes in, what comes out**
  - *The idea:* A prediction model learns a link between inputs and an answer. The inputs are called **features** and are stored in a table named `X`; the answer to predict is the **target**, stored in `y`. Every row is one example: one applicant's features and that applicant's real outcome.

  - *See it on real data:*

Column	Role	Why
`Age_Years_Count`, `Annual_Income_USD`, `Credit_Utilization_Pct`, `Education_Tier`	features (`X`)	facts known when the application arrives
`Defaulted_Binary`	target (`y`)	the outcome we want to predict
`Client_ID`	neither	a label; it says nothing about risk

  - *Common mistake:* leaving the target inside `X`. The model would "predict" default by looking at the answer itself, score perfectly in testing, and be useless on new applicants, who have no answer yet.
  - *More examples:*

Code	Result	What it tells you
`X.shape`	`(12, 4)`	12 applicants, 4 features
`y.value_counts()`	`0`: 9, `1`: 3	3 of 12 applicants defaulted
`y.mean()`	`0.25`	the default rate: the mean of a 0/1 column is the share of 1s
`X.isnull().sum()`	`1` in income, `1` in utilization	the gaps to fill before training

  - *Remember:* `X` = what you know in advance; `y` = what you want to predict.

- **7.2 Filling gaps: imputation**
  - *The idea:* Models cannot learn from empty cells. Imputation fills each gap with a typical value of its column. The **median** (the middle value) is a safe default, because a few extreme values do not pull it, unlike the average (mean).
  - *See it on real data:* the income column, with `CLI-707`'s income missing.

Statistic of the 11 known incomes	Value
mean	`115727.27`
median	`110000.0`
smallest and largest	`58000.0` and `185000.0`

    The 3 high incomes of the Doctorate applicants pull the mean up; the median stays in the middle.
  - *Common mistake:* deleting every row with a gap. With 12 rows, dropping 2 loses one-sixth of the data. Fill gaps when you can; drop rows only when too much of the row is missing.
  - *More examples:*

Code	Result	What it tells you
`df["Credit_Utilization_Pct"].median()`	`36.5`	the value that would fill `CLI-709`'s gap, using all rows
`SimpleImputer(strategy="median").fit(X_train).statistics_`	`[38.0, 96000.0, 33.15, 0.0, 0.0]`	the medians learned from the 9 training rows only
`SimpleImputer(strategy="mean")`	fills with averages	another option for columns without extreme values

The medians from all rows and from the training rows differ (`36.5` vs `33.15`), because they are computed from different applicants. Section 7.5 explains why the training version is the correct one.

- *Remember:* fill gaps with the median, learned from the training rows.

- **7.3 Text categories become 0/1 columns: one-hot encoding**
  - *The idea:* A model computes with numbers, so the text in `Education_Tier` must become numbers. Writing `Bachelor = 1`, `Master = 2`, `Doctorate = 3` would wrongly tell the model that a Doctorate is "three times" a Bachelor. One-hot encoding instead makes one 0/1 column per category. With `drop_first=True`, the first category gets no column: it is the row with `0` everywhere.
  - *See it on real data:*

`Education_Tier`	`Education_Tier_Doctorate`	`Education_Tier_Master`
`Bachelor`	`0`	`0`
`Master`	`0`	`1`
`Doctorate`	`1`	`0`

Categories are put in alphabetical order, so `Bachelor` comes first and is the one dropped.

  - *Common mistake:* a category appearing in new data that was not in the training data, such as `"Associate"`. `pd.get_dummies()` would create a column the model has never seen. Exercise 3 shows a tool that handles this safely.
  - *More examples:*

Code	Result	What it tells you
`df["Education_Tier"].value_counts()`	Master 5, Bachelor 4, Doctorate 3	how many 1s each new column will have
`pd.get_dummies(X, columns=["Education_Tier"], drop_first=True).shape`	`(12, 5)`	1 text column replaced by 2 number columns
the same with `drop_first=False`	`(12, 6)`	3 columns, one per category

  - *Remember:* one 0/1 column per category, never 1-2-3 codes.

- **7.4 Putting numbers on one scale: standardization**
  - *The idea:* Income runs from `58,000` to `185,000`; utilization from `15.2` to `62.1`. Some models compare rows by distance and would treat a difference of 1,000 dollars as far bigger than a difference of 10 percentage points, only because of the units. Standardization rewrites each value as "how many standard deviations above or below the average", so all columns use the same scale. The standard deviation measures how spread out a column is.
  - *See it on real data:* age, average `42.5` years, standard deviation `10.66` years (all 12 applicants).

Applicant age	Calculation	Standardized
`28`	`(28 - 42.5) / 10.66`	`-1.36`
`45`	`(45 - 42.5) / 10.66`	`0.23`
`63`	`(63 - 42.5) / 10.66`	`1.92`

    After standardization, every column has an average of `0` and a standard deviation of `1`.
  - *More examples:* income, average `115727.27`, standard deviation `43668.36` (the 11 known values).

Value	Calculation	Standardized
income `58000`	`(58000 - 115727.27) / 43668.36`	`-1.32`: the lowest income
income `110000`	`(110000 - 115727.27) / 43668.36`	`-0.13`: just below average
income `185000`	`(185000 - 115727.27) / 43668.36`	`1.59`: well above average
`df["Annual_Income_USD"].std()`	`45799.76`	pandas divides by n - 1 by default; the scaler uses n (`43668.36`)

  - *Remember:* standardized value = (value - average) / standard deviation.
- **7.5 Honest testing: split first, then learn**
  - *The idea:* The test set plays the role of next month's new applicants: the model and its preparation steps must not see it during training. So the order is fixed: split the rows first; `fit` the imputer and the scaler on the training rows only; then `transform` both sets with what was learned. Using information from the test set during training is called **data leakage**, and it makes a model look better than it really is.
  - *See it on real data:* `train_test_split(X, y, test_size=0.25, random_state=42, stratify=y)`.

Setting	Effect here
`test_size=0.25`	3 of 12 rows go to the test set: `CLI-704`, `CLI-710`, `CLI-708`
`stratify=y`	2 defaults in training and 1 in test: the same 25% rate in both

Setting	Effect here
`random_state=42`	the same split every time you run the script
`random_state=7` instead	a different test set: `CLI-710`, `CLI-705`, `CLI-701`

  - *Common mistake:* `scaler.fit_transform(X_test)`. The scaler then learns the test rows' own average, and the test applicants all look "average", even though they are older and earn more than the training applicants. Exercise 2 shows the numbers.
  - *More examples:*

Setting	Train / test rows	Defaults in train / test
`test_size=0.25, stratify=y` (ours)	9 / 3	2 / 1
`test_size=0.25`, no `stratify`	9 / 3	2 / 1 here, but not guaranteed; the test set is `CLI-710`, `CLI-709`, `CLI-700`
`test_size=0.5, stratify=y`	6 / 6	2 / 1: 3 defaults cannot split evenly
scaler fitted on train, test averages	`0.74`, `0.73`, `0.57`, …	correct: the test applicants differ
scaler fitted on test (leakage)	all `0.0`	the difference has disappeared

  - *Remember:* split, fit on train, transform both.

8. Development Workflow in Google Antigravity IDE

- `[1]` File Explorer: create your script with your name and the session code (for example, `Reacher_S32_demo_1.py`), and check the dataset is in `data/`.
- `[2]` Code Editor: write a few lines, run, repeat. Small steps keep errors small.
- `[3]` AI Agent Chat: on an error, capture the traceback (Python's error report) with `Win + Shift + S` (macOS: `Cmd + Shift + 4`), paste it with `Ctrl + V`, and ask why it failed.
- `[4]` Terminal: run `python` followed by your script name. Return code `0` means the script finished with no errors. First time only: `pip install scikit-learn`.

9. Hands-On Coding Micro-Examples

- Every example has three parts: **1. Input**, **2. Process**, and **3. Output**. Name each file with your own name and the session code, for example `Reacher_S32_demo_1.py` (pattern: `YourName_S32_demo_1.py`).
- *Micro-Example 1 (Features, Target, and Gaps):* `YourName_S32_demo_1.py`

```
# 1. Input: load the applicants
import pandas as pd
```

```
df = pd.read_csv("data/session_03_02_dataset.csv")

# 2. Process: inputs (X) and the answer to predict (y)
X = df.drop(columns=["Client_ID", "Defaulted_Binary"])
y = df["Defaulted_Binary"]

# 3. Output: size, outcome counts, and missing values
print("X shape (rows, columns):", X.shape)
print("Defaults (1) and repaid (0):\n", y.value_counts())
print("Missing values per column:\n", X.isnull().sum())
```

  - *How it works, line by line:*
    - `df.drop(columns=["Client_ID", "Defaulted_Binary"])` returns a copy of the table without the ID and the answer: the features `X` (section 7.1). The original `df` is unchanged.
    - `df["Defaulted_Binary"]` picks the answer column: the target `y`.
    - `y.value_counts()` counts each answer; `X.isnull().sum()` counts the gaps per column.
  - *Read the output:*

```
X shape (rows, columns): (12, 4)
Defaults (1) and repaid (0):
 Defaulted_Binary
0    9
1    3
Missing values per column:
 Age_Years_Count           0
Annual_Income_USD         1
Credit_Utilization_Pct    1
Education_Tier            0
```

    - 4 features for 12 applicants.
    - 3 defaults out of 12: a 25% default rate, the first number a credit manager asks for.
    - One income and one utilization value are missing: they must be filled before training (section 7.2).

- *Quick Check 1 (`value_counts(normalize=True)`):*
  - New tool: `.value_counts(normalize=True)` gives each value's share instead of its count. Why: the default rate is the first number a credit manager asks for.
  - Task: add one line to part **3. Output**: `print(y.value_counts(_____=True))`
  - Check: `0` has `0.75` and `1` has `0.25`: a 25% default rate.

- *Micro-Example 2 (One-Hot Encoding):* `YourName_S32_demo_2.py`

```
# 1. Input: load the applicants and keep the features
import pandas as pd
df = pd.read_csv("data/session_03_02_dataset.csv")
X = df.drop(columns=["Client_ID", "Defaulted_Binary"])
```

```
# 2. Process: one 0/1 column per education level, except the first
X_encoded = pd.get_dummies(X, columns=["Education_Tier"],
                           drop_first=True, dtype=float)

# 3. Output: the new column names and two rows
print(X_encoded.columns.tolist())
print(X_encoded.head(2))
```

  - *How it works, line by line:*
    - `pd.get_dummies(X, columns=["Education_Tier"], drop_first=True, dtype=float)` replaces the text column with 0/1 columns (section 7.3). `dtype=float` writes them as `0.0` and `1.0`.
    - `X_encoded.columns.tolist()` lists the new column names.
    - `.head(2)` shows the first two applicants, a Bachelor and a Master.
  - *Read the output:*

```
['Age_Years_Count', 'Annual_Income_USD', 'Credit_Utilization_Pct',
 ↪  'Education_Tier_Doctorate', 'Education_Tier_Master']
   Age_Years_Count  ...  Education_Tier_Master
0               28  ...                    0.0
1               45  ...                    1.0
```

    - `Education_Tier` is gone; two number columns took its place. There is no `Education_Tier_Bachelor`: a Bachelor is the row with `0.0` in both.
    - Applicant `0` (`CLI-700`) is a Bachelor (`0.0` in the Master column); applicant `1` (`CLI-701`) is a Master (`1.0`).

- *Quick Check 2 (`drop_first`):*
  - New tool: the `drop_first` setting. `True` leaves out the first category; `False` keeps a column for every category. Why: shows what "the first category is implied" means.
  - Task: in part **2. Process**, change `drop_first=True` to `drop_first=_____`.
  - Check: a third column, `Education_Tier_Bachelor`, appears, and each row has exactly one `1.0` among the three education columns. Change it back to `True` afterwards.

- *Micro-Example 3 (Split First, Then Fill and Scale):* `YourName_S32_demo_3.py`

```
# 1. Input: load, encode, and split before learning anything
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
df = pd.read_csv("data/session_03_02_dataset.csv")
X = pd.get_dummies(df.drop(columns=["Client_ID", "Defaulted_Binary"]),
                   columns=["Education_Tier"], drop_first=True, dtype=float)
y = df["Defaulted_Binary"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42, stratify=y
```

```
)

# 2. Process: learn medians and scales from the training rows only
imputer = SimpleImputer(strategy="median")
X_train_filled = imputer.fit_transform(X_train)
X_test_filled = imputer.transform(X_test)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_filled)
X_test_scaled = scaler.transform(X_test_filled)

# 3. Output: sizes and the medians used to fill the gaps
print("Train rows:", len(X_train), "| Test rows:", len(X_test))
print("Medians learned from training rows:",
 ↪  imputer.statistics_.round(2).tolist())
```

  - *How it works, line by line:*
    - Part 1 encodes the text column, then splits the rows **before** anything is learned (section 7.5).
    - `SimpleImputer(strategy="median")` creates an empty gap-filler. `.fit_transform(X_train)` learns the training medians and fills the training gaps; `.transform(X_test)` fills the test gaps with those same training medians.
    - `StandardScaler()` works the same way: it learns each column's training average and standard deviation, then applies them to both sets (section 7.4).
    - `imputer.statistics_` holds the medians it learned; the trailing _ marks a value learned during `fit`.
  - *Read the output:*

```
Train rows: 9 | Test rows: 3
Medians learned from training rows: [38.0, 96000.0, 33.15, 0.0, 0.0]
```

    - 9 applicants for training and 3 kept aside for the final test.
    - The medians are, in column order: age `38.0`, income `96000.0`, utilization `33.15`, and `0.0` for both education columns. Any gap, in either set, is filled with these values.

- *Quick Check 3 (`scaler.mean_`):*
  - New tool: `scaler.mean_` holds the column averages the scaler learned from the training rows. Why: shows exactly what "fit on the training set" means.
  - Task: add one line to part **3. Output**: `print("Averages learned by the scaler:", scaler._____.round(2).tolist())`
  - Check: the first three values are `40.44` (age), `106444.44` (income), and `35.82` (utilization).

10. Official Python Documentation & References

- Scikit-Learn Preprocessing Guide: StandardScaler and OneHotEncoder.
- Scikit-Learn Imputation of Missing Values: SimpleImputer.
- Scikit-Learn Model Selection: train_test_split.
- Scikit-Learn: Common Pitfalls (Data Leakage).

11. Exercise 1: Guided Practice - Credit Data Preparation Pipeline

- *Step 1: Open Script:* In Google Antigravity IDE, create a new script named `YourName_S32_preprocessing.py`, replacing `YourName` with your own name (for example, `Reacher_S32_preprocessing.py`).
- *Step 2: Build the Full Preparation Sequence:* Type the following code:

```
# 1. Input: load, encode, and split
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
df = pd.read_csv("data/session_03_02_dataset.csv")
X = pd.get_dummies(df.drop(columns=["Client_ID", "Defaulted_Binary"]),
                   columns=["Education_Tier"], drop_first=True, dtype=float)
y = df["Defaulted_Binary"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42, stratify=y
)

# 2. Process: fill, then scale; fit on training rows only
imputer = SimpleImputer(strategy="median")
scaler = StandardScaler()
X_train_scaled = pd.DataFrame(
    scaler.fit_transform(imputer.fit_transform(X_train)), columns=X.columns
)
X_test_scaled = pd.DataFrame(
    scaler.transform(imputer.transform(X_test)), columns=X.columns
)

# 3. Output: check the training columns
print("Training averages:\n", X_train_scaled.mean().round(2))
print("Training spread:\n", X_train_scaled.std(ddof=0).round(2))
print("Defaults in train:", y_train.sum(), "| in test:", y_test.sum())
```

  - `.std(ddof=0)` uses the same standard deviation formula as the scaler. pandas' default formula gives a slightly larger value (`1.06` here) for small tables.

- *Step 3: Validate the Scaling:* Every training average should be `0.0` (a printed `-0.0` is the same as `0.0`) and every spread `1.0`.
- *Step 4: Execute in Terminal:* Run `python YourName_S32_preprocessing.py` in Panel `[4]` and confirm return code `0`.
- *Success Check:* 5 columns, each with average `0.0` and spread `1.0`; 2 defaults in the training set and `1` in the test set, thanks to `stratify=y`.

12. Exercise 2: Applied Practice - Feature Engineering and a Leakage Check

- *Coding Challenge:* In `YourName_S32_practice.py`, add one engineered feature, then see with your own eyes what data leakage hides.
- *Task 1 (Engineer a Feature):* Add `High_Utilization_Flag` = `1` when `Credit_Utilization_Pct` is above `50`, else `0`: `(df["Credit_Utilization_Pct"] > 50).astype(int)`. `.astype(int)` turns `True` / `False` into `1` / `0`. Print `pd.crosstab(df["High_Utilization_Flag"], df["Defaulted_Binary"])`.
- *Task 2 (Read the NotFittedError & AI Diagnosis):* Starting from Micro-Example 3, create a new `StandardScaler()` and call `.transform(X_test_filled)` on it before any `.fit()`. The script stops with `NotFittedError: This StandardScaler instance is not fitted yet.` Capture the traceback with `Win + Shift + S`, paste it into Panel `[3]` (AI Agent Chat) with `Ctrl + V`, and ask what "fitted" means.
- *Task 3 (See the Leakage):* Scale the test rows two ways and print the column averages of each, rounded to 2 decimals:
  - Correct: `scaler.transform(X_test_filled)`, with `scaler` fitted on the training rows
  - Leaky: `StandardScaler().fit_transform(X_test_filled)`
  - `array.mean(axis=0)` gives the average of each column of a Scikit-Learn result.
- *Success Check:*
  - Task 1: the flag splits the applicants perfectly: all 3 flagged applicants defaulted and all 9 others repaid. With only 12 rows, treat a perfect pattern as a warning sign, not as proof.
  - Task 3: the correct averages are `0.74`, `0.73`, `0.57`, `0.27`, `0.71`, showing that the test applicants are older, earn more, and use more credit than the training applicants. The leaky averages are all `0.0`: the difference has disappeared.

13. Exercise 3: Advanced Practice - A Reusable Preparation Pipeline

- *Goal:* Build one preparation object that handles numbers and text in one step, can be fitted on training rows and reused on any new applicant, and does not crash on a category it has never seen.
- *What you need:*
  - File: `data/session_03_02_dataset.csv` (described in **5. Lab Practice Dataset**).
  - Script name: `YourName_S32_challenge.py`.
  - Tools you already know: `train_test_split` (same settings as Micro-Example 3), `SimpleImputer`, `StandardScaler`, and `.fit()` / `.transform()`.
- *Requirements (follow them in order):*
  1. Split the raw features (before any encoding) and the target with `test_size=0.25`, `random_state=42`, and `stratify=y`.
  2. Define `numeric_cols = ["Age_Years_Count", "Annual_Income_USD", "Credit_Utilization_Pct"]` and `text_cols = ["Education_Tier"]`.
  3. Build a `Pipeline` for the numeric columns: fill with the median, then standardize.
  4. Combine it with a `OneHotEncoder(handle_unknown="ignore")` for the text column in one `ColumnTransformer`.

5. Fit the combined object on the training rows only. Print the output column names, then the transformed test rows, rounded to 2 decimals.
6. Transform one new applicant: age `36`, income missing (`None`), utilization `57.0`, education `"Associate"`. Print the result, rounded to 2 decimals.
7. In a comment, explain what happened to the missing income and to the unseen education level.

- *New tools for this exercise:*
  - `from sklearn.pipeline import Pipeline`: `Pipeline([("impute", SimpleImputer(strategy="median")), ("scale", StandardScaler())])` runs its steps in order, and fits them in order.
  - `from sklearn.compose import ColumnTransformer`: sends each group of columns to its own tool and joins the results side by side, for example:

    ```
    ColumnTransformer([
        ("num", numeric_pipeline, numeric_cols),
        ("cat", OneHotEncoder(handle_unknown="ignore"), text_cols),
    ])
    ```

  - `from sklearn.preprocessing import OneHotEncoder`: the Scikit-Learn version of one-hot encoding. `handle_unknown="ignore"` writes `0` in every category column for a value it never saw, instead of raising an error.
  - `.get_feature_names_out()` returns the names of the output columns.
  - A one-row DataFrame for the new applicant: `pd.DataFrame({"Age_Years_Count": [36], "Annual_Income_USD": [None], "Credit_Utilization_Pct": [57.0], "Education_Tier": ["Associate"]})`.
- *Hints:*
  - Wrap a result in `pd.DataFrame(result, columns=prep.get_feature_names_out())` to read it with column names.
  - Without `handle_unknown="ignore"`, the new applicant raises `ValueError: Found unknown categories ['Associate']`. Try it once to see why the setting matters.
- *Self-Check (expected results):*
  - Output columns (6): `num__Age_Years_Count`, `num__Annual_Income_USD`, `num__Credit_Utilization_Pct`, `cat__Education_Tier_Bachelor`, `cat__Education_Tier_Doctorate`, `cat__Education_Tier_Master`.
  - Transformed test rows: shape `(3, 6)`; the second test row is the Doctorate applicant with `1.41`, `1.63`, `1.49`.
  - New applicant: `-0.4`, `-0.25`, `1.4`, then `0.0`, `0.0`, `0.0`. The missing income was filled with the training median (`96000.0`) before scaling, and the unseen `"Associate"` level became all zeros instead of crashing the script. Business meaning: the same object can now prepare every future application in one call, exactly as it prepared the training data.