02_data_preprocessing

August 21, 2025

0.0.1 Data Preprocessing

In this file, we begin the **data preprocessing** steps before modeling.
The main objectives are:
1. **Create the target variable** — identify whether a contract is a **lapse (1)** or **not lapse (0)**.
2. **Ensure all features are numeric (except transaction_description)** — categorical/text fields will be encoded or dropped as needed. transaction_description will be handled in next file using llm. 3. **Balance the dataset for training** — since lapse contracts are rare, special care is taken to address class imbalance.

0.0.2 Handling Missing Values

Our dataset is **highly imbalanced** (very few lapse contracts compared to non-lapse).
Because of this, we are **not imputing missing values** (to avoid introducing bias).
Instead, we are **removing rows with missing data**.

Even though we are not filling missing values in this project, some common approaches include:
- **Mean/Median Imputation**: replacing missing values with the average or median of the column.
- **Forward Fill**: filling with the value from the previous row.
- **Backward Fill**: filling with the value from the next row.
- **Interpolation**: averaging between the previous and next data points.
- **Conditional Imputation**: filling based on values from other columns (e.g., group-wise averages).
- **Domain-Specific Rules**: using business logic to fill missing entries (e.g., defaults, constants).

---

[1]:
```
# Standard library
import logging

# Third-party / Scientific stack
import numpy as np
import pandas as pd
from scipy.stats import chi2_contingency

# Machine learning / feature engineering
from imblearn.under_sampling import NearMiss
from sklearn.feature_selection import mutual_info_classif
```

```python
# Ignore warning
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```python
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(message)s"
)
```

Import collected data Previously collected and merged **Compustat** and **USAspending** datasets were saved as
`contract_data.csv` for downstream analysis.

[3]:
```python
df_contract = pd.read_csv('collected_data.csv')
df_contract
```

[3]:
```
                           contract_transaction_unique_key  \
0             9700_9700_SPE2DV22F78C7_0_SPE2DV17D4001_0
1             9700_9700_SPE2D622F523K_0_SPE2DE18D0010_0
2             9700_9700_SPE2DM22FN9VG_0_SPE2DM20D9503_0
3             9700_9700_SPE2DM22FY4Z0_0_SPE2DM20D2504_0
4             9700_9700_SPE2DV22FALX8_0_SPE2DV17D6030_0
…                                                     …
1711202        7008_-NONE-_70Z08724PSTPL0002_0_-NONE-_0
1711203       1443_-NONE-_140P2123P0036_P00001_-NONE-_0
1711204       12D0_-NONE-_12FPC122P0005_P00001_-NONE-_0
1711205       9700_-NONE-_W9127N21P0149_P00002_-NONE-_0
1711206  7008_-NONE-_70Z08724PSTPL0002_P00001_-NONE-_0

                                  contract_award_unique_key      award_id_piid  \
0         CONT_AWD_SPE2DV22F78C7_9700_SPE2DV17D4001_9700      SPE2DV22F78C7
1         CONT_AWD_SPE2D622F523K_9700_SPE2DE18D0010_9700      SPE2D622F523K
2         CONT_AWD_SPE2DM22FN9VG_9700_SPE2DM20D9503_9700      SPE2DM22FN9VG
3         CONT_AWD_SPE2DM22FY4Z0_9700_SPE2DM20D2504_9700      SPE2DM22FY4Z0
4         CONT_AWD_SPE2DV22FALX8_9700_SPE2DV17D6030_9700      SPE2DV22FALX8
…                                                     …                …
1711202    CONT_AWD_70Z08724PSTPL0002_7008_-NONE-_-NONE-  70Z08724PSTPL0002
1711203        CONT_AWD_140P2123P0036_1443_-NONE-_-NONE-      140P2123P0036
1711204        CONT_AWD_12FPC122P0005_12D0_-NONE-_-NONE-      12FPC122P0005
1711205        CONT_AWD_W9127N21P0149_9700_-NONE-_-NONE-      W9127N21P0149
1711206    CONT_AWD_70Z08724PSTPL0002_7008_-NONE-_-NONE-  70Z08724PSTPL0002

         federal_action_obligation  total_dollars_obligated  \
0                            38.60                    38.60
1                           109.60                   109.60
2                          6033.03                  6033.03
```

```
3                         98.80                   98.80
4                        255.75                  255.75
…                           …                      …
1711202                  104.00                  104.00
1711203                    0.00                19913.00
1711204                 -256.00                 1988.25
1711205                -5000.00                 4000.00
1711206                    0.00                  104.00

         current_total_value_of_award  potential_total_value_of_award  \
0                               38.60                           38.60
1                              109.60                          109.60
2                             6033.03                         6033.03
3                               98.80                           98.80
4                              255.75                          255.75
…                                 …                              …
1711202                        104.00                          104.00
1711203                      19913.00                        19913.00
1711204                       1988.25                         1988.25
1711205                       4000.00                         4000.00
1711206                        104.00                          104.00

        action_date  action_date_fiscal_year period_of_performance_start_date  \
0        2021-11-10                     2022                       2021-11-10
1        2021-11-10                     2022                       2021-11-10
2        2021-11-10                     2022                       2021-11-10
3        2021-11-10                     2022                       2021-11-10
4        2021-11-10                     2022                       2021-11-10
…             …                      …                                  …
1711202  2024-07-16                     2024                       2024-07-16
1711203  2024-07-11                     2024                       2023-07-11
1711204  2024-06-11                     2024                       2021-11-10
1711205  2024-08-29                     2024                       2021-09-30
1711206  2024-08-28                     2024                       2024-07-16

         …                company_name         at      sale      revt  \
0        …            OWENS & MINOR INC   3536.551  9785.315  9785.315
1        …            PATTERSON COS INC   2741.630  6499.405  6499.405
2        …            OWENS & MINOR INC   3536.551  9785.315  9785.315
3        …            OWENS & MINOR INC   3536.551  9785.315  9785.315
4        …            OWENS & MINOR INC   3536.551  9785.315  9785.315
…       …                           …         …        …        …
1711202  …  FIRST AMERICAN FINANCIAL CP  16802.800  5998.100  5998.100
1711203  …  FIRST AMERICAN FINANCIAL CP  16802.800  5998.100  5998.100
1711204  …  FIRST AMERICAN FINANCIAL CP  16802.800  5998.100  5998.100
1711205  …  FIRST AMERICAN FINANCIAL CP  16802.800  5998.100  5998.100
1711206  …  FIRST AMERICAN FINANCIAL CP  16802.800  5998.100  5998.100
```

```
              ib          lt        ceq    oancf  xrd       cogs
0        221.589    2598.050    938.501  124.177  0.0   8272.086
1        203.210    1698.995   1041.676 -980.994  NaN   5079.132
2        221.589    2598.050    938.501  124.177  0.0   8272.086
3        221.589    2598.050    938.501  124.177  0.0   8272.086
4        221.589    2598.050    938.501  124.177  0.0   8272.086
…          …         …          …        …     …      …
1711202  216.800   11940.000   4848.100  354.300  NaN   5408.100
1711203  216.800   11940.000   4848.100  354.300  NaN   5408.100
1711204  216.800   11940.000   4848.100  354.300  NaN   5408.100
1711205  216.800   11940.000   4848.100  354.300  NaN   5408.100
1711206  216.800   11940.000   4848.100  354.300  NaN   5408.100

[1711207 rows x 55 columns]
```

Filter contracts by country For this project, we only consider contracts issued **to and performed within the United States**.

[4]:
```python
# Keep only rows where both country columns == 'UNITED STATES'
df_contract = df_contract[
    (df_contract["primary_place_of_performance_country_name"] == "UNITED␣
↪STATES") &
    (df_contract["recipient_country_name"] == "UNITED STATES")
].copy()
```

0.0.3 Identify non-numeric columns

As we need primarily **numeric data** for modeling, let's check which columns contain string (categorical or text) values.

[5]:
```python
print("\n=== String columns (with unique value counts, descending) ===")
string_cols  = df_contract.select_dtypes(include=["object", "string"]).columns.
↪tolist()
string_uniques = {c: df_contract[c].nunique() for c in string_cols}
for col, n in sorted(string_uniques.items(), key=lambda x: x[1], reverse=True):
    print(f"{col}: {n} unique")
```

```
=== String columns (with unique value counts, descending) ===
contract_transaction_unique_key: 1690718 unique
contract_award_unique_key: 1501695 unique
award_id_piid: 1498716 unique
transaction_description: 1478733 unique
period_of_performance_potential_end_date: 9137 unique
period_of_performance_current_end_date: 5278 unique
period_of_performance_start_date: 4601 unique
product_or_service_code: 1511 unique
```

```
company_name: 1446 unique
action_date: 1096 unique
naics_description: 240 unique
sic_desc: 150 unique
type_of_contract_pricing: 15 unique
extent_competed: 9 unique
award_type: 4 unique
undefinitized_action: 3 unique
performance_based_service_acquisition: 3 unique
government_furnished_property: 2 unique
veteran_owned_business: 2 unique
woman_owned_business: 2 unique
minority_owned_business: 2 unique
contracting_officers_determination_of_business_size: 2 unique
us_state_government: 2 unique
us_local_government: 2 unique
us_tribal_government: 2 unique
educational_institution: 2 unique
hospital_flag: 2 unique
foreign_owned: 2 unique
for_profit_organization: 2 unique
nonprofit_organization: 2 unique
the_ability_one_program: 2 unique
small_disadvantaged_business: 2 unique
recipient_country_name: 1 unique
primary_place_of_performance_country_name: 1 unique
foreign_government: 1 unique
historically_black_college: 1 unique
tribal_college: 1 unique
```

[6]:

```
# Drop the unwanted columns
drop_cols = [
    "contract_transaction_unique_key",
    "contract_award_unique_key",
    "award_id_piid",
    "name_usasp",
    "name_usasp_norm",
    "naics_description",
    "sic_desc",
    "historically_black_college",
    "tribal_college",
    "primary_place_of_performance_country_name",
    "recipient_country_name",
    "action_date",
    "company_name",
    "foreign_government",
    "naics_code"
```

```
]
df_contract = df_contract.drop(columns=[c for c in drop_cols if c in␣
↪df_contract.columns], errors="ignore")
```

[7]:
```
# Convert datetime like columns to datetime
date_cols = [
    "period_of_performance_current_end_date",
    "period_of_performance_start_date",
    "period_of_performance_potential_end_date"
]
for col in date_cols:
    if col in df_contract.columns:
        df_contract[col] = pd.to_datetime(df_contract[col], errors="coerce")
```

[8]:
```
print("\n=== String columns (with unique value counts, descending) ===")
string_cols  = df_contract.select_dtypes(include=["object", "string"]).columns.
↪tolist()
string_uniques = {c: df_contract[c].nunique() for c in string_cols}
for col, n in sorted(string_uniques.items(), key=lambda x: x[1], reverse=True):
    print(f"{col}: {n} unique")
```

```
=== String columns (with unique value counts, descending) ===
transaction_description: 1478733 unique
product_or_service_code: 1511 unique
type_of_contract_pricing: 15 unique
extent_competed: 9 unique
award_type: 4 unique
undefinitized_action: 3 unique
performance_based_service_acquisition: 3 unique
government_furnished_property: 2 unique
veteran_owned_business: 2 unique
woman_owned_business: 2 unique
minority_owned_business: 2 unique
contracting_officers_determination_of_business_size: 2 unique
us_state_government: 2 unique
us_local_government: 2 unique
us_tribal_government: 2 unique
educational_institution: 2 unique
hospital_flag: 2 unique
foreign_owned: 2 unique
for_profit_organization: 2 unique
nonprofit_organization: 2 unique
the_ability_one_program: 2 unique
small_disadvantaged_business: 2 unique
```

Our dataset contains several **string columns** that require special handling before modeling.

- For `transaction_description`, we will use an **LLM** to classify descriptions into **themes**.

- For other categorical columns (e.g., `type_of_contract_pricing`, `extent_competed`), we can map categories into **numeric codes** (0 … *n_unique*).

- For very small cardinality (binary or near-binary flags), we can treat them as **0/1 indicators**.

- An exception is `product_or_service_code` which, given its interpretive nature, might require more careful feature engineering rather than raw numeric mapping.

[9]:
```python
# Let's see how many values are int and how many alphanumeric in␣
↪product_or_service_code
codes = df_contract["product_or_service_code"].astype(str)

# Boolean mask: True if the entire code is only digits
is_numeric = codes.str.fullmatch(r"\d+")

# Count rows
rows_numeric = is_numeric.sum()
rows_alphanumeric = (~is_numeric).sum()

print(f"Rows with only numeric codes: {rows_numeric}")
print(f"Rows with alphanumeric codes: {rows_alphanumeric}")
```

```
Rows with only numeric codes: 1481393
Rows with alphanumeric codes: 209329
```

[10]:
```python
# Check if PSC can be simplified by using first 3 digits (reduce unique values)
psc_2digit = (
    df_contract["product_or_service_code"]
    .dropna()
    .astype(str)
    .str.extract(r"^(\d{3})")[0]
)

print("Unique 3-digit PSC codes:", psc_2digit.nunique())
print("Sample codes:", psc_2digit.dropna().unique()[:20])
```

```
Unique 3-digit PSC codes: 364
Sample codes: ['651' '664' '852' '650' '653' '652' '654' '655' '691' '663' '890'
'891'
 '792' '894' '611' '312' '109' '151' '172' '531']
```

Product/Service Code (PSC) Feature Engineering We will use frequency encoding to convert PSC into numeric values.

Reason: PSC has high cardinality (many unique codes).

Frequency-based encoding reduces dimensionality while preserving signal, since more common PSC codes get higher values and rare ones lower.

```
[11]: # Extract first 3 digits (numeric only)
df_contract["psc_3digit"] = (
    df_contract["product_or_service_code"]
    .astype(str)
    .str.extract(r"^(\d{3})")[0]
)

# Compute frequencies of each PSC code
freq_map = df_contract["psc_3digit"].value_counts().to_dict()

# Map frequencies back into dataframe
df_contract["psc_3digit_freq"] = df_contract["psc_3digit"].map(freq_map)

# Drop intermediate and original PSC columns
df_contract = df_contract.drop(columns=["psc_3digit",␣
 ↪"product_or_service_code"], errors="ignore")
```

0.0.4 Drop sparse binary flags

Since these features have extremely few `t` values compared to `f`, they provide little predictive power. We will remove them from the dataset.

hospital_flag
educational_institution

```
[12]: # --- Count unique values and their presence ---
for col in ["hospital_flag", "educational_institution"]:
    print(f"\n=== {col} ===")
    print(df_contract[col].value_counts(dropna=False))
    print(f"Unique values: {df_contract[col].nunique(dropna=False)}")
```

```
=== hospital_flag ===
hospital_flag
f    1690340
t        382
Name: count, dtype: int64
Unique values: 2

=== educational_institution ===
educational_institution
f    1690617
t        105
Name: count, dtype: int64
Unique values: 2
```

```
[13]: # Drop unnecessary columns
df_contract = df_contract.drop(
```

```python
    ["hospital_flag", "educational_institution"],
    axis=1,
    errors="ignore"
)
```

Filter to Federal Government Contracts For this project, we only consider contracts issued directly by the **federal government**.
Thus, we remove contracts associated with other levels of government and drop their columns:

1. Keep only rows where `us_tribal_government == 'f'`, then drop the column.
2. Keep only rows where `us_state_government == 'f'`, then drop the column.
3. Keep only rows where `us_local_government == 'f'`, then drop the column.

After applying these filters, the dataset contains only **federal-level contracts**.

[14]:
```python
# Keep only rows where us_tribal_government == 'f', then drop column
if "us_tribal_government" in df_contract.columns:
    df_contract = df_contract[df_contract["us_tribal_government"] == "f"]
    df_contract = df_contract.drop("us_tribal_government", axis=1)

# Keep only rows where us_state_government == 'f', then drop column
if "us_state_government" in df_contract.columns:
    df_contract = df_contract[df_contract["us_state_government"] == "f"]
    df_contract = df_contract.drop("us_state_government", axis=1)

# Keep only rows where us_local_government == 'f', then drop column
if "us_local_government" in df_contract.columns:
    df_contract = df_contract[df_contract["us_local_government"] == "f"]
    df_contract = df_contract.drop("us_local_government", axis=1)

print("Final shape after filtering:", df_contract.shape)
```

```
Final shape after filtering: (1690495, 37)
```

Encode Categorical Variables To prepare the dataset for modeling, we convert categorical string fields into numeric form.
We use **factorization**, which assigns each unique value in a column to an integer label (0…N-1).

[15]:
```python
# Columns to encode
cat_cols = [
    "type_of_contract_pricing",
    "extent_competed",
    "award_type",
    "undefinitized_action",
    "performance_based_service_acquisition",
    "the_ability_one_program",
```

```
    "nonprofit_organization",
    "for_profit_organization",
    "foreign_owned",
    "contracting_officers_determination_of_business_size",
    "minority_owned_business",
    "woman_owned_business",
    "veteran_owned_business",
    "government_furnished_property",
    "small_disadvantaged_business"
]

# Factorize each column into 0..N-1 integers
for col in cat_cols:
    df_contract[col], uniques = pd.factorize(df_contract[col])
    print(f"{col}: {len(uniques)} unique -> encoded as 0..{len(uniques)-1}")
```

```
type_of_contract_pricing: 15 unique -> encoded as 0..14
extent_competed: 9 unique -> encoded as 0..8
award_type: 4 unique -> encoded as 0..3
undefinitized_action: 3 unique -> encoded as 0..2
performance_based_service_acquisition: 3 unique -> encoded as 0..2
the_ability_one_program: 2 unique -> encoded as 0..1
nonprofit_organization: 2 unique -> encoded as 0..1
for_profit_organization: 2 unique -> encoded as 0..1
foreign_owned: 2 unique -> encoded as 0..1
contracting_officers_determination_of_business_size: 2 unique -> encoded as 0..1
minority_owned_business: 2 unique -> encoded as 0..1
woman_owned_business: 2 unique -> encoded as 0..1
veteran_owned_business: 2 unique -> encoded as 0..1
government_furnished_property: 2 unique -> encoded as 0..1
small_disadvantaged_business: 2 unique -> encoded as 0..1
```

0.0.5 Check if we have any string columns

Before proceeding, we need to confirm that all remaining features are numeric.
The only exception is **`transaction_description`**, which we will later transform into **themes using an LLM**.
After that step, we will convert the themes into numeric values.

For now, we ensure there are **no other string columns** left in the dataset.

[16]:
```
print("\n=== String columns (with unique value counts, descending) ===")
string_cols  = df_contract.select_dtypes(include=["object", "string"]).columns.
 ↪tolist()
string_uniques = {c: df_contract[c].nunique() for c in string_cols}
for col, n in sorted(string_uniques.items(), key=lambda x: x[1], reverse=True):
    print(f"{col}: {n} unique")
```

```
=== String columns (with unique value counts, descending) ===
transaction_description: 1478636 unique
```

0.0.6 Creating the Target Variable (lapse_flag)

To model whether a government contract **lapses or not**, we create a binary target variable `lapse_flag`.
The labeling logic is based on comparing the **current end date** and the **potential end date** of the contract, relative to the fiscal year end (September 30, 2024).

Rationale behind the rules:

1. **Contract still active (label = NaN)**
   - If the **potential end date** extends **beyond FY2024**, we cannot know yet if it will lapse, so we leave it unlabeled (`NaN`).
2. **Missing dates (label = NaN)**
   - If either the **potential end** or **current end date** is missing, we cannot determine status, so we skip labeling.
3. **Completed contracts (label = 0)**
   - If the **gap between potential end and current end** is small (30 days), it suggests the contract ran to completion.
4. **Lapsed contracts (label = 1)**
   - If the **gap is large (180 days)**, it indicates the contract ended much earlier than planned → treated as lapsed.
5. **Ambiguous cases (label = NaN)**
   - If the gap is between 30 and 180 days, we consider it uncertain and do not assign a label.

[17]:
```python
check_date = pd.Timestamp("2024-09-30")  # End of FY2024

def label_contract(row):
    """Assign lapse_flag: 0=completed, 1=lapsed, NaN=uncertain/active."""

    pot_end = row["period_of_performance_potential_end_date"]
    cur_end = row["period_of_performance_current_end_date"]

    # If potential end is beyond FY2024 → still active
    if pd.notna(pot_end) and pot_end > check_date:
        return np.nan

    # If either date missing → cannot classify
    if pd.isna(pot_end) or pd.isna(cur_end):
        return np.nan

    # Compute gap between potential and current end
    gap_days = (pot_end - cur_end).days

    # Completed
    if gap_days <= 30:
        return 0
```

```
    # Lapsed
    if gap_days >= 180:
        return 1

    # Ambiguous
    return np.nan

# Apply row-wise labeling
df_contract["lapse_flag"] = df_contract.apply(label_contract, axis=1)

# Summary
print("Counts of labeled contracts:")
print(df_contract["lapse_flag"].value_counts(dropna=False))
```

```
Counts of labeled contracts:
lapse_flag
0.0    1573490
NaN     104737
1.0      12268
Name: count, dtype: int64
```

We only **keep contracts with a definitive outcome (completed = 0, lapsed = 1)** and drop rows with NaN labels coz we don't know if those contract (NaN) will become success or get lapse.

[18]:
```
# Remove NaN labels (keep only 0 and 1) ---
df_contract = df_contract.dropna(subset=["lapse_flag"]).copy()
df_contract["lapse_flag"] = df_contract["lapse_flag"].astype(int)  # cast to int
```

[19]:
```
# Remove datetime columns used for target variable creation

# Identify datetime columns
datetime_cols = df_contract.select_dtypes(include=["datetime64[ns]",␣
 ↪"datetimetz"]).columns.tolist()
print("Datetime columns to remove:", datetime_cols)

# Drop them
df_contract = df_contract.drop(columns=datetime_cols)
```

```
Datetime columns to remove: ['period_of_performance_start_date',
'period_of_performance_current_end_date',
'period_of_performance_potential_end_date']
```

We also **remove rows with negative federal_action_obligation**. A negative value typically reflects adjustments, cancellations, or de-obligations of funds rather than a true spending commitment.

[20]:
```
# Remove rows with negative federal_action_obligation
df_contract = df_contract[df_contract["federal_action_obligation"] >= 0].copy()
```

Check NaN

[21]:
```
# Replace inf with NaN
df_contract = df_contract.replace([np.inf, -np.inf], np.nan)

# Replace problematic string patterns like "Onan", "nan", "NaN"
df_contract = df_contract.replace(
    to_replace=[r'^\s*0?nan\s*$', r'^\s*NaN\s*$', r'^\s*nan\s*$'],
    value=np.nan,
    regex=True
)

# --- Count NaNs per column ---
nan_counts = df_contract.isna().sum()
nan_counts = nan_counts[nan_counts > 0].sort_values(ascending=False)

print("NaN counts per column:")
print(nan_counts)
```

```
NaN counts per column:
xrd                303577
psc_3digit_freq    116534
lt                     60
ceq                    25
oancf                  21
sale                   15
revt                   15
ib                     15
cogs                   15
dtype: int64
```

Analysis of PSC (Product/Service Codes) We observe that the `psc_3digit_freq` column has a large number of NaN values.
Before deciding whether to keep or drop it, we first examine if it has any relationship with our target variable (`lapse_flag`).
If certain PSC groups are strongly associated with higher or lower lapse rates, this feature could provide useful predictive signal despite missing values.

[22]:
```
# Build contingency table between PSC frequency and lapse flag
contingency = pd.crosstab(df_contract["psc_3digit_freq"],␣
↪df_contract["lapse_flag"])

# Run chi-square test of independence
chi2, p, dof, expected = chi2_contingency(contingency)

# Print test statistic and p-value
print("Chi-square:", chi2, "p-value:", p)
```

```
Chi-square: 52831.199551776794 p-value: 0.0
```

[23]:
```
# Compute lapse rates grouped by PSC frequency
rates = (
    df_contract.groupby("psc_3digit_freq")["lapse_flag"]
    .mean()  # mean gives proportion of lapse_flag = 1
    .sort_values(ascending=False)  # sort by highest lapse rate
)

# Display top 20 PSC groups with highest lapse rate
print(rates.head(20))
```

```
psc_3digit_freq
173.0     0.395161
57.0      0.166667
58.0      0.159420
38.0      0.133333
9.0       0.128205
77.0      0.127660
10.0      0.111111
88.0      0.103448
36.0      0.100775
459.0     0.100173
109.0     0.100000
193.0     0.077519
3129.0    0.066555
21.0      0.048780
2664.0    0.045732
87.0      0.045455
55.0      0.040000
1922.0    0.039932
1315.0    0.039414
99.0      0.037736
Name: lapse_flag, dtype: float64
```

The analysis shows that **PSC 3-digit codes have a relationship with contract lapse**, making them a potentially important predictor.

However, the dataset is **highly imbalanced** — we have only about 12,200 lapsed contracts compared to over a million non-lapsed contracts. Since we will be **calling an LLM to classify transaction_description**, we must limit the dataset size to a few thousand samples to **manage API costs**. For final model training, we will therefore keep an **equal number of lapsed and non-lapsed contracts**, which also ensures **balanced training**.

For now, we will drop rows with NaN values, so that only valid and usable data remain for modeling.

[24]:
```
# Drop rows with any NaN
df_contract = df_contract.dropna(axis=0, how="any")

#df_contract
```

Awarding agency code vs Funding agency code

[25]:
```
# Check if awarding and funding agency codes match
same_all = (df_contract["awarding_agency_code"] ==␣
↪df_contract["funding_agency_code"]).all()

if same_all:
    print("All awarding_agency_code match funding_agency_code")
else:
    mismatches = (df_contract["awarding_agency_code"] !=␣
↪df_contract["funding_agency_code"]).sum()
    print(f"Mismatches found: {mismatches}")
```

```
Mismatches found: 426
```

> **Note:** Among millions of rows, only 488 cases showed mismatches between `awarding_agency_code` and `funding_agency_code`. Since the overlap is overwhelmingly high, we drop the `awarding_agency_code` column to avoid redundancy.

[26]:
```
# Drop redundant column
df_contract = df_contract.drop(columns=["awarding_agency_code"],␣
↪errors="ignore")
```

Check feature engineering

[27]:
```
print("Remaining columns:", len(df_contract.columns))

# --- Print string (object) columns and their NaN counts ---
str_cols = df_contract.select_dtypes(include="object").columns
nan_counts = df_contract[str_cols].isna().sum()

print("String columns and NaN counts:\n", nan_counts)
print("Remaining columns:", len(df_contract.columns))
```

```
Remaining columns: 34
String columns and NaN counts:
 transaction_description    0
dtype: int64
Remaining columns: 34
```

Feature Importance using Mutual Information Before model training, it is useful to identify which features provide the most information about the target variable.
We apply **mutual information (MI)** between each feature and the lapse flag (`lapse_flag`) to capture both linear and non-linear relationships.

Categorical columns are first encoded into integer codes so that MI can be computed.
The resulting MI scores are sorted to highlight the top predictors, which helps guide feature selection and domain interpretation.

[28]:
```
# Separate X and y
X = df_contract.drop(columns=["lapse_flag"])
y = df_contract["lapse_flag"]

# Encode categorical cols temporarily
X_enc = X.copy()
for col in X_enc.select_dtypes(include="object").columns:
    X_enc[col] = X_enc[col].astype("category").cat.codes

# Compute mutual information
mi = mutual_info_classif(X_enc, y, discrete_features="auto")

# Show sorted importance
mi_series = pd.Series(mi, index=X_enc.columns).sort_values(ascending=False)
print(mi_series.head(20))
```

```
funding_agency_code                                    0.258985
sic4                                                   0.148613
psc_3digit_freq                                        0.114182
action_date_fiscal_year                                0.109065
lt                                                     0.041600
at                                                     0.041444
revt                                                   0.041135
cogs                                                   0.040732
sale                                                   0.040634
oancf                                                  0.031575
ib                                                     0.028629
ceq                                                    0.007549
type_of_contract_pricing                               0.006489
transaction_description                                0.004803
xrd                                                    0.003895
potential_total_value_of_award                         0.003494
current_total_value_of_award                           0.003334
total_dollars_obligated                                0.003244
contracting_officers_determination_of_business_size    0.002939
federal_action_obligation                              0.002123
dtype: float64
```

0.0.7 Addressing Class Imbalance

The dataset is **highly imbalanced**, with far fewer lapsed contracts compared to non-lapsed contracts.

To build a reliable model, we will make the number of lapsed and non-lapsed contracts equal for each year. This step is important because without balancing, the model would be biased toward predicting the majority (non-lapsed) class, leading to poor recall on the minority (lapsed) class, which is our main focus.

[29]:
```
# Year-wise summary
print("Year-wise lapse_flag counts")
print(df_contract.groupby("action_date_fiscal_year")["lapse_flag"].
 ↪value_counts(dropna=False))
```

```
Year-wise lapse_flag counts
action_date_fiscal_year  lapse_flag
2022                     0             470928
                         1                770
2023                     0             390675
                         1                397
2024                     0             313468
                         1                149
Name: count, dtype: int64
```

Balancing Strategy: Hybrid NearMiss + Random Sampling The dataset is highly imbalanced, with only a few thousand lapsed contracts compared to millions of non-lapsed ones. To address this, we apply a **year-wise balancing strategy**:

- **Keep all lapsed contracts** (`lapse_flag = 1`) for each year.
- For each year's non-lapsed contracts (`lapse_flag = 0`), select an **equal number** to match the lapses.
- The non-lapsed contracts are chosen using a **hybrid method**:
  - **50% via NearMiss** → closest non-lapsed to lapses (hard negatives near the boundary).
  - **50% via random sampling** → from the remaining pool to ensure diversity.

This ensures that:
- Each year has a balanced **1:1 ratio** of lapses and non-lapses.
- Negatives are both **informative** (NearMiss) and **representative** (random).
- The final dataset is **shuffled and ready** for modeling.

[30]:
```
def hybrid_nearmiss_random_by_year(
    df,
    year_col="action_date_fiscal_year",
    target_col="lapse_flag",
    random_state=42,
):
    """
    Balance dataset year-wise using a hybrid of NearMiss and random sampling.

    - Keeps all lapse_flag = 1 rows.
    - Selects equal number of non-lapse (0) rows.
    - 50% of non-lapse chosen via NearMiss, 50% via random sampling.

    Args:
```

```
        df (pd.DataFrame): Input dataframe.
        year_col (str): Column name for year grouping.
        target_col (str): Target column (0/1).
        random_state (int): Random seed.

    Returns:
        pd.DataFrame: Balanced dataframe.
    """
    rng = np.random.RandomState(random_state)
    parts = []

    for yr, d in df.groupby(year_col, sort=True):
        pos = d[d[target_col] == 1]
        neg = d[d[target_col] == 0]
        n_pos, n_neg = len(pos), len(neg)

        if n_pos == 0:
            continue
        if n_neg == 0:
            parts.append(pos)
            continue

        # Numeric features for distance
        num_cols = d.select_dtypes(include=["number"]).columns.difference(
            [target_col]
        )

        if not len(num_cols):
            neg_random = neg.sample(
                n=min(n_neg, n_pos),
                random_state=random_state,
                replace=False,
            )
            parts.append(pd.concat([pos, neg_random], axis=0))
            logging.info(
                "[%s] Fallback random: lapses=%d, non-lapses=%d",
                yr,
                len(pos),
                len(neg_random),
            )
            continue

        # Impute numeric data
        X_imp = (
            d[num_cols]
            .replace([np.inf, -np.inf], np.nan)
            .fillna(d[num_cols].median(numeric_only=True))
```

```
            .fillna(0.0)
        )
        y = d[target_col].astype(int)

        # Apply NearMiss
        nm = NearMiss(version=1, n_neighbors=3)
        X_res, y_res = nm.fit_resample(X_imp, y)
        selected_idx = d.index[nm.sample_indices_]
        nm_neg = d.loc[selected_idx][d[target_col] == 0]

        half_nearmiss = n_pos // 2
        nm_neg_keep = nm_neg.iloc[:half_nearmiss]

        # Random negatives
        remaining_neg = neg.drop(index=nm_neg_keep.index, errors="ignore")
        n_random_needed = n_pos - len(nm_neg_keep)
        neg_rand_keep = (
            remaining_neg.sample(
                n=min(n_random_needed, len(remaining_neg)),
                random_state=random_state,
                replace=False,
            )
            if n_random_needed > 0
            else remaining_neg.iloc[0:0]
        )

        neg_keep = pd.concat([nm_neg_keep, neg_rand_keep], axis=0)

        # Top-up if still short
        if len(neg_keep) < n_pos:
            residual_pool = neg.drop(index=neg_keep.index, errors="ignore")
            extra = residual_pool.sample(
                n=min(n_pos - len(neg_keep), len(residual_pool)),
                random_state=random_state,
                replace=False,
            )
            neg_keep = pd.concat([neg_keep, extra], axis=0)

        year_balanced = pd.concat([pos, neg_keep], axis=0)
        parts.append(year_balanced)

        cnt = year_balanced[target_col].value_counts().to_dict()
        logging.info(
            "[%s] lapses=%d, non-lapses=%d (NearMiss=%d, Random=%d)",
            yr,
            cnt.get(1, 0),
            cnt.get(0, 0),
```

```
            len(nm_neg_keep),
            len(neg_rand_keep),
        )

    if not parts:
        return pd.DataFrame(columns=df.columns)

    out = pd.concat(parts, ignore_index=True)
    out = out.sample(frac=1.0, random_state=random_state).reset_index(drop=True)

    return out
```

[31]:
```
df_hybrid = hybrid_nearmiss_random_by_year(
    df_contract,
    year_col="action_date_fiscal_year",
    target_col="lapse_flag",
    random_state=42,
)
```

```
2025-08-21 17:46:55,252 | INFO | [2022] lapses=770, non-lapses=770
(NearMiss=385, Random=385)
2025-08-21 17:46:55,707 | INFO | [2023] lapses=397, non-lapses=397
(NearMiss=198, Random=199)
2025-08-21 17:46:56,010 | INFO | [2024] lapses=149, non-lapses=149 (NearMiss=74,
Random=75)
```

[32]:
```
# Year-wise summary
print("Year-wise lapse_flag counts")
print(df_hybrid.groupby("action_date_fiscal_year")["lapse_flag"].
 ↪value_counts(dropna=False))
```

```
Year-wise lapse_flag counts
action_date_fiscal_year  lapse_flag
2022                     0             770
                         1             770
2023                     0             397
                         1             397
2024                     0             149
                         1             149
Name: count, dtype: int64
```

[33]:
```
df_hybrid.to_csv('fea_eng_basic.csv', index=False)
```

0.0.8 Baseline Modeling Setup

To get an initial sense of predictive power, we will now try a simple model.
For data preparation, we split the dataset by year:

- **Training data:** 2022 and 2023

- **Test data:** 2024

This split simulates a realistic scenario where past contract information is used to predict outcomes for future contracts. The goal here is not final optimization, but rather to validate whether the prepared features contain predictive signal.

[34]:
```
# Train/Test Split by Year
train = df_hybrid[df_hybrid["action_date_fiscal_year"].isin([2022, 2023])].
 ↪copy()
test  = df_hybrid[df_hybrid["action_date_fiscal_year"] == 2024].copy()
```

Before modeling, we drop the following columns:

- **transaction_description**: this column is free text (string) and requires LLM-based processing, which is not part of this baseline model.

- **action_date_fiscal_year**: this has already been used for splitting train/test sets, so including it would cause data leakage.

[35]:
```
# Drop columns not needed for ML
drop_cols = ["action_date_fiscal_year", "transaction_description"]

train = train.drop(columns=drop_cols, errors="ignore")
test  = test.drop(columns=drop_cols, errors="ignore")
```

[36]:
```
# Libraries
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, roc_auc_score, precision_recall_curve,␣
 ↪auc
```

[37]:
```
# Separate features and target
X_train = train.drop(columns=["lapse_flag"])
y_train = train["lapse_flag"]

X_test  = test.drop(columns=["lapse_flag"])
y_test  = test["lapse_flag"]
```

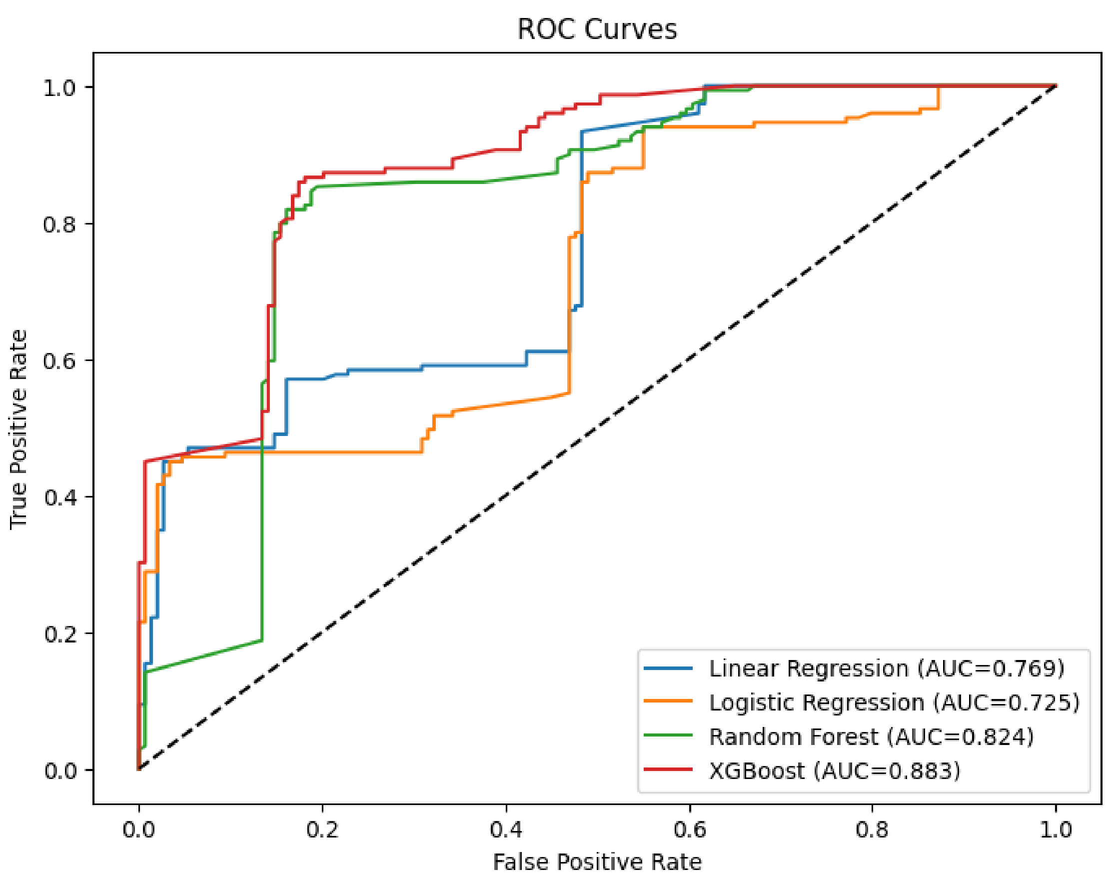
[38]:
```
# 1. Linear Regression
lr = LinearRegression()
lr.fit(X_train, y_train)
y_pred_lr = (lr.predict(X_test) > 0.5).astype(int)
print("Linear Regression Accuracy:", accuracy_score(y_test, y_pred_lr))
```

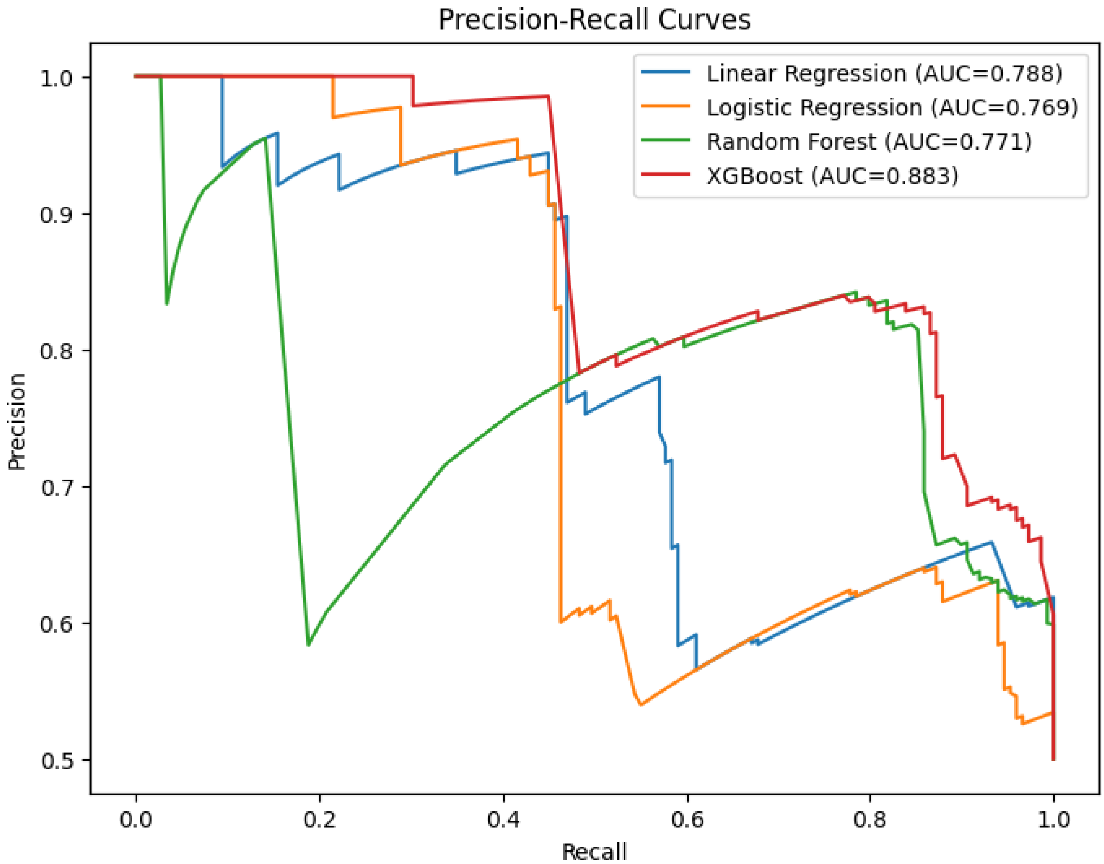
```python
# 2. Logistic Regression
logr = LogisticRegression(max_iter=1000)
logr.fit(X_train, y_train)
y_pred_logr = logr.predict(X_test)
print("Logistic Regression Accuracy:", accuracy_score(y_test, y_pred_logr))

# 3. Random Forest
rf = RandomForestClassifier(n_estimators=200, random_state=42)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))

# 4. XGBoost
xgb = XGBClassifier(use_label_encoder=False, eval_metric="logloss",␣
 ↪random_state=42)
xgb.fit(X_train, y_train)
y_pred_xgb = xgb.predict(X_test)
print("XGBoost Accuracy:", accuracy_score(y_test, y_pred_xgb))
```

```
Linear Regression Accuracy: 0.7013422818791947
Logistic Regression Accuracy: 0.6845637583892618
Random Forest Accuracy: 0.8221476510067114
XGBoost Accuracy: 0.8322147651006712
```

[39]:
```python
# Store models and predictions
models = {
    "Linear Regression": (lr, y_pred_lr),
    "Logistic Regression": (logr, y_pred_logr),
    "Random Forest": (rf, y_pred_rf),
    "XGBoost": (xgb, y_pred_xgb),
}

# --- ROC Curves ---
plt.figure(figsize=(8, 6))
for name, (model, y_pred) in models.items():
    if hasattr(model, "predict_proba"):
        y_proba = model.predict_proba(X_test)[:, 1]
    else:
        # For Linear Regression, already thresholded
        y_proba = lr.predict(X_test)

    fpr, tpr, _ = roc_curve(y_test, y_proba)
    auc_score = roc_auc_score(y_test, y_proba)
    plt.plot(fpr, tpr, label=f"{name} (AUC={auc_score:.3f})")

plt.plot([0, 1], [0, 1], "k--")
plt.xlabel("False Positive Rate")
```

```python
plt.ylabel("True Positive Rate")
plt.title("ROC Curves")
plt.legend()
plt.show()

# Precision-Recall Curves
plt.figure(figsize=(8, 6))
for name, (model, y_pred) in models.items():
    if hasattr(model, "predict_proba"):
        y_proba = model.predict_proba(X_test)[:, 1]
    else:
        y_proba = lr.predict(X_test)

    precision, recall, _ = precision_recall_curve(y_test, y_proba)
    pr_auc = auc(recall, precision)
    plt.plot(recall, precision, label=f"{name} (AUC={pr_auc:.3f})")

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curves")
plt.legend()
plt.show()
```





Tree-based algorithms (e.g., Random Forest, XGBoost) tend to work better in our setup because we use simple integer encodings (0 ... n_unique) instead of one-hot encoding. Linear models misinterpret these encodings as ordered numeric values, while tree-based methods handle them naturally by splitting on thresholds, effectively learning category partitions without needing high-dimensional one-hot vectors.