1_pipeline_preprocessing

July 3, 2025

[ ]:

```python

```

[1]:

```python
"""
    Complete Scikit-Learn Preprocessing Pipeline for ISBSG Data
    ============================================================

    This module provides a comprehensive preprocessing pipeline that handles:
    1. Data loading and initial cleaning
    2. Column name standardization
    3. Missing value handling
    4. Semicolon-separated value processing
    5. One-hot encoding for categorical variables
    6. Multi-label binarization for multi-value columns
    7. Feature selection and filtering
    8. Data validation and export

    Based on the preprocessing steps from the provided notebooks.
   """
```

[1]:

```
'\n    Complete Scikit-Learn Preprocessing Pipeline for ISBSG Data\n
============================================================\n\n    This module
provides a comprehensive preprocessing pipeline that handles:\n    1. Data
loading and initial cleaning\n    2. Column name standardization\n    3. Missing
value handling\n    4. Semicolon-separated value processing\n    5. One-hot
encoding for categorical variables\n    6. Multi-label binarization for multi-
value columns\n    7. Feature selection and filtering\n    8. Data validation
and export\n\n    Based on the preprocessing steps from the provided
notebooks.\n   '
```

[2]:

```python
# === Imports ===

import pandas as pd
import numpy as np
import re
from pathlib import Path
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MultiLabelBinarizer, StandardScaler
```

```
from sklearn.compose import ColumnTransformer
import joblib
import os
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
import warnings
from collections import Counter, defaultdict
from typing import List, Dict, Any, Tuple
warnings.filterwarnings('ignore')
```

[3]:
```
# Configuration
DATA_FOLDER = "../data"
CONFIG_FOLDER = "../config"
SAMPLE_FILE = "ISBSG2016R1_1_financial_industry_seed.xlsx"
FULL_FILE = "ISBSG2016R1_1_full_dataset.xlsx"
TARGET_COL = "project_prf_normalised_work_effort"  # be careful about case␣
↪sensitive
```

[4]:
```
from collections import Counter
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np

def analyze_high_cardinality_multivalue(df, column, separator=';'):
    """
    Analyze high-cardinality multi-value columns to choose best strategy
    """
    print(f"=== ANALYSIS FOR HIGH-CARDINALITY COLUMN: '{column}' ===\n")

    # Basic statistics
    non_null_data = df[column].dropna().astype(str)
    split_values = non_null_data.apply(lambda x: [v.strip() for v in x.
↪split(separator) if v.strip()])

    # Get all unique values
    all_values = []
    for values_list in split_values:
        all_values.extend(values_list)

    value_counts = Counter(all_values)
    unique_values = list(value_counts.keys())

    print(f"Total unique values: {len(unique_values)}")
    print(f"Total value occurrences: {len(all_values)}")
    print(f"Average values per row: {len(all_values) / len(split_values):.2f}")
```

```python
    # Show most common values
    print(f"\nTop 15 most common values:")
    for value, count in value_counts.most_common(15):
        percentage = (count / len(non_null_data)) * 100
        print(f"  '{value}': {count} times ({percentage:.1f}% of rows)")

    # Show distribution of value frequencies
    frequency_dist = Counter(value_counts.values())
    print(f"\nFrequency distribution:")
    for freq, count in sorted(frequency_dist.items(), reverse=True)[:10]:
        print(f"  {count} values appear {freq} time(s)")

    # Values per row distribution
    values_per_row = split_values.apply(len)
    print(f"\nValues per row:")
    print(f"  Min: {values_per_row.min()}")
    print(f"  Max: {values_per_row.max()}")
    print(f"  Mean: {values_per_row.mean():.2f}")
    print(f"  Median: {values_per_row.median():.2f}")

    return value_counts, unique_values


def handle_high_cardinality_multivalue(df, multi_value_columns, separator=';', strategy='top_k', preserve_original=None, **kwargs):
    """
    Handle high-cardinality multi-value columns with various strategies

    Parameters:
    -----------
    strategy options:
    - 'top_k': Keep only top K most frequent values (k=kwargs['k'])
    - 'frequency_threshold': Keep values that appear in at least X% of rows (threshold=kwargs['threshold'])
    - 'tfidf': Use TF-IDF vectorization with dimensionality reduction (n_components=kwargs['n_components'])
    - 'count_features': Simple counting features (count, unique_count, most_common)
    - 'embedding': Create category embeddings (requires pre-trained embeddings)
    """

    df_processed = df.copy()
    new_columns_mapping = {}
    preserve_original = preserve_original or []

    for col in multi_value_columns:
```

```
        if col not in df.columns:
            continue

        print(f"\nProcessing high-cardinality column '{col}' with strategy '{strategy}'...")

        # Clean and split values
        split_values = df[col].fillna('').astype(str).apply(
            lambda x: [val.strip() for val in x.split(separator) if val.strip()]
        )

        # Get value counts
        all_values = []
        for values_list in split_values:
            all_values.extend(values_list)
        value_counts = Counter(all_values)

        if strategy == 'top_k':
            k = kwargs.get('k', 20)  # Default to top 20
            top_values = [val for val, count in value_counts.most_common(k)]

            new_col_names = []
            for value in top_values:
                new_col_name = f"{col}_top_{value}".replace(' ', '_').replace('-', '_')
                df_processed[new_col_name] = split_values.apply(lambda x: 1 if value in x else 0)
                new_col_names.append(new_col_name)

            # Add "other" category for remaining values
            other_col_name = f"{col}_other"
            df_processed[other_col_name] = split_values.apply(
                lambda x: 1 if any(val not in top_values for val in x) else 0
            )
            new_col_names.append(other_col_name)

            new_columns_mapping[col] = new_col_names
            print(f"  Created {len(new_col_names)} columns (top {k} + other)")

        elif strategy == 'frequency_threshold':
            threshold = kwargs.get('threshold', 0.05)  # Default 5%
            min_occurrences = int(len(df) * threshold)

            frequent_values = [val for val, count in value_counts.items() if count >= min_occurrences]

            new_col_names = []
```

```
            for value in frequent_values:
                new_col_name = f"{col}_freq_{value}".replace(' ', '_').
↪replace('-', '_')
                df_processed[new_col_name] = split_values.apply(lambda x: 1 if␣
↪value in x else 0)
                new_col_names.append(new_col_name)

            # Add rare category
            rare_col_name = f"{col}_rare"
            df_processed[rare_col_name] = split_values.apply(
                lambda x: 1 if any(val not in frequent_values for val in x)␣
↪else 0
            )
            new_col_names.append(rare_col_name)

            new_columns_mapping[col] = new_col_names
            print(f"  Created {len(new_col_names)} columns␣
↪({len(frequent_values)} frequent + rare)")

        elif strategy == 'count_features':
            # Create aggregate features instead of individual columns
            new_col_names = []

            # Total count of values
            count_col = f"{col}_count"
            df_processed[count_col] = split_values.apply(len)
            new_col_names.append(count_col)

            # Unique count (in case of duplicates)
            unique_count_col = f"{col}_unique_count"
            df_processed[unique_count_col] = split_values.apply(lambda x:␣
↪len(set(x)))
            new_col_names.append(unique_count_col)

            # Most common value in the dataset appears in this row
            most_common_value = value_counts.most_common(1)[0][0] if␣
↪value_counts else None
            if most_common_value:
                most_common_col = f"{col}_has_most_common"
                df_processed[most_common_col] = split_values.apply(lambda x: 1␣
↪if most_common_value in x else 0)
                new_col_names.append(most_common_col)

            # Average frequency of values in this row
            avg_freq_col = f"{col}_avg_frequency"
            df_processed[avg_freq_col] = split_values.apply(
```

```
            lambda x: np.mean([value_counts[val] for val in x]) if x else 0
        )
        new_col_names.append(avg_freq_col)

        new_columns_mapping[col] = new_col_names
        print(f"  Created {len(new_col_names)} aggregate feature columns")

    elif strategy == 'tfidf':
        n_components = kwargs.get('n_components', 10)  # Default to 10␣
↪components

        # Convert to text format for TF-IDF
        text_data = split_values.apply(lambda x: ' '.join(x))

        # Apply TF-IDF
        tfidf = TfidfVectorizer(max_features=100, stop_words=None)
        tfidf_matrix = tfidf.fit_transform(text_data)

        # Reduce dimensionality
        pca = PCA(n_components=n_components)
        tfidf_reduced = pca.fit_transform(tfidf_matrix.toarray())

        # Create new columns
        new_col_names = []
        for i in range(n_components):
            new_col_name = f"{col}_tfidf_comp_{i+1}"
            df_processed[new_col_name] = tfidf_reduced[:, i]
            new_col_names.append(new_col_name)

        new_columns_mapping[col] = new_col_names
        print(f"  Created {len(new_col_names)} TF-IDF component columns")
        print(f"  Explained variance ratio: {pca.
↪explained_variance_ratio_}")

    elif strategy == 'hierarchical':
        # Group similar values into higher-level categories
        # This requires domain knowledge - example implementation
        hierarchy = kwargs.get('hierarchy', {})  # Dictionary mapping␣
↪values to categories

        if not hierarchy:
            print("  Warning: No hierarchy provided for hierarchical␣
↪strategy")
            continue

        # Create columns for each high-level category
        categories = set(hierarchy.values())
```

```
            new_col_names = []

            for category in categories:
                category_values = [val for val, cat in hierarchy.items() if cat == category]
                new_col_name = f"{col}_category_{category}".replace(' ', '_')
                df_processed[new_col_name] = split_values.apply(
                    lambda x: 1 if any(val in category_values for val in x) else 0
                )
                new_col_names.append(new_col_name)

            new_columns_mapping[col] = new_col_names
            print(f"  Created {len(new_col_names)} hierarchical category columns")

        # FIXED: Only remove original column if NOT in preserve list
        if col not in preserve_original:
            df_processed = df_processed.drop(columns=[col])
            print(f"  Removed original column '{col}'")
        else:
            print(f"   Preserved original column '{col}' (in exclude list)")

    return df_processed, new_columns_mapping


def recommend_strategy(df, column, separator=';'):
    """
    Recommend the best strategy based on data characteristics
    """
    value_counts, unique_values = analyze_high_cardinality_multivalue(df, column, separator)

    total_unique = len(unique_values)
    total_rows = len(df[column].dropna())

    print(f"\n=== STRATEGY RECOMMENDATIONS FOR '{column}' ===")

    if total_unique > 100:
        print("  VERY HIGH CARDINALITY (100+ unique values)")
        print("Recommended strategies:")
        print("1. 'count_features' - Create aggregate features (safest)")
        print("2. 'top_k' with k=15-25 - Keep only most important values")
        print("3. 'tfidf' with n_components=5-10 - If values have semantic meaning")

    elif total_unique > 50:
```

```
        print("  HIGH CARDINALITY (50+ unique values)")
        print("Recommended strategies:")
        print("1. 'top_k' with k=20-30 - Keep most frequent values")
        print("2. 'frequency_threshold' with threshold=0.02-0.05")
        print("3. 'count_features' - If you want aggregate information")

    else:
        print("  MODERATE CARDINALITY (<50 unique values)")
        print("Recommended strategies:")
        print("1. 'frequency_threshold' with threshold=0.01")
        print("2. 'top_k' with k=30-40")
        print("3. Binary encoding might be acceptable")

    # Check frequency distribution
    freq_values = list(value_counts.values())
    if max(freq_values) / min(freq_values) > 100:
        print("\n   HIGHLY SKEWED DISTRIBUTION detected")
        print("   Consider 'frequency_threshold' or 'top_k' strategies")
```

[5]:
```
def validate_multivalue_processing(df_original, df_processed, original_column,␣
↪new_columns, separator=';', strategy='top_k'):
    """
    Comprehensive validation of multi-value categorical processing

    Parameters:
    -----------
    df_original : pd.DataFrame
        Original dataset before processing
    df_processed : pd.DataFrame
        Processed dataset after handling multi-value columns
    original_column : str
        Name of original multi-value column
    new_columns : list
        List of new column names created from the original column
    separator : str
        Separator used in original data
    strategy : str
        Strategy used for processing
    """

    print(f"=== VALIDATION REPORT FOR COLUMN '{original_column}' ===\n")

    # 1. BASIC CHECKS
    print("1. BASIC INTEGRITY CHECKS")
    print("-" * 40)

    # Check row count consistency
```

```python
    original_rows = len(df_original)
    processed_rows = len(df_processed)
    print(f" Row count: {original_rows} → {processed_rows} {' SAME' if original_rows == processed_rows else '  DIFFERENT'}")

    # Check if original column was removed
    original_removed = original_column not in df_processed.columns
    print(f" Original column removed: {' YES' if original_removed else ' NO'}")

    # Check if new columns exist
    new_cols_exist = all(col in df_processed.columns for col in new_columns)
    print(f" New columns created: {' YES' if new_cols_exist else ' NO'} ({len(new_columns)} columns)")

    if not new_cols_exist:
        missing_cols = [col for col in new_columns if col not in df_processed.columns]
        print(f"  Missing columns: {missing_cols}")
        return False

    # 2. DATA CONSISTENCY CHECKS
    print(f"\n2. DATA CONSISTENCY CHECKS")
    print("-" * 40)

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.split(separator) if v.strip()])

    # Get all unique values from original
    all_original_values = set()
    for values_list in split_original:
        all_original_values.update(values_list)
    all_original_values = sorted([v for v in all_original_values if v and v != 'nan'])

    print(f"Original unique values: {len(all_original_values)}")

    if strategy == 'top_k':
        # Validate top-k strategy
        validate_top_k_strategy(df_original, df_processed, original_column, new_columns, separator)
    elif strategy == 'count_features':
        validate_count_features_strategy(df_original, df_processed, original_column, new_columns, separator)
```

```python
    elif strategy == 'frequency_threshold':
        validate_frequency_threshold_strategy(df_original, df_processed,␣
↪original_column, new_columns, separator)

    # 3. SAMPLE VALIDATION
    print(f"\n3. SAMPLE-BY-SAMPLE VALIDATION")
    print("-" * 40)
    validate_sample_rows(df_original, df_processed, original_column,␣
↪new_columns, separator, n_samples=5)

    # 4. STATISTICAL VALIDATION
    print(f"\n4. STATISTICAL VALIDATION")
    print("-" * 40)
    validate_statistics(df_original, df_processed, original_column,␣
↪new_columns, separator)

    # 5. INFORMATION LOSS ASSESSMENT
    print(f"\n5. INFORMATION LOSS ASSESSMENT")
    print("-" * 40)
    assess_information_loss(df_original, df_processed, original_column,␣
↪new_columns, separator)

    return True


def validate_top_k_strategy(df_original, df_processed, original_column,␣
↪new_columns, separator, k=None):
    """Validate top-k strategy specifically"""

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.
↪split(separator) if v.strip()])

    # Get value counts
    all_values = []
    for values_list in split_original:
        all_values.extend(values_list)
    value_counts = Counter(all_values)

    # Determine k if not provided
    if k is None:
        # Exclude "other" column to determine k
        non_other_cols = [col for col in new_columns if not col.
↪endswith('_other')]
        k = len(non_other_cols)
```

```
    top_k_values = [val for val, count in value_counts.most_common(k)]
    print(f"Top {k} values: {top_k_values[:5]}{'...' if len(top_k_values) > 5␣
↪else ''}")

    # Check each top-k column
    for col in new_columns:
        if col.endswith('_other'):
            # Validate "other" column
            validate_other_column(df_original, df_processed, original_column,␣
↪col, top_k_values, separator)
        else:
            # Extract the value name from column name
            value_name = col.replace(f"{original_column}_top_", "").
↪replace(f"{original_column}_", "")
            validate_binary_column(df_original, df_processed, original_column,␣
↪col, value_name, separator)


def validate_binary_column(df_original, df_processed, original_column,␣
↪new_column, value_name, separator):
    """Validate a single binary column"""

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.
↪split(separator) if v.strip()])

    # Expected values: 1 if value_name in the list, 0 otherwise
    expected = split_original.apply(lambda x: 1 if value_name in x else 0)
    actual = df_processed[new_column]

    # Compare
    matches = (expected == actual).sum()
    total = len(expected)
    match_rate = matches / total * 100

    print(f"  '{new_column}': {matches}/{total} matches ({match_rate:.1f}%)")

    if match_rate < 100:
        mismatches = df_original.loc[expected != actual, original_column].
↪head(3)
        print(f"    Sample mismatches: {list(mismatches)}")
```

```
def validate_other_column(df_original, df_processed, original_column,␣
↪other_column, top_values, separator):
    """Validate the 'other' category column"""

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.
↪split(separator) if v.strip()])

    # Expected: 1 if any value is NOT in top_values, 0 if all values are in␣
↪top_values
    expected = split_original.apply(lambda x: 1 if any(val not in top_values␣
↪for val in x) else 0)
    actual = df_processed[other_column]

    matches = (expected == actual).sum()
    total = len(expected)
    match_rate = matches / total * 100

    print(f"  '{other_column}': {matches}/{total} matches ({match_rate:.1f}%)")


def validate_count_features_strategy(df_original, df_processed,␣
↪original_column, new_columns, separator):
    """Validate count features strategy"""

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.
↪split(separator) if v.strip()])

    for col in new_columns:
        if col.endswith('_count'):
            # Validate total count
            expected = split_original.apply(len)
            actual = df_processed[col]
            matches = (expected == actual).sum()
            print(f"  '{col}': {matches}/{len(expected)} matches ({matches/
↪len(expected)*100:.1f}%)")

        elif col.endswith('_unique_count'):
            # Validate unique count
            expected = split_original.apply(lambda x: len(set(x)))
            actual = df_processed[col]
            matches = (expected == actual).sum()
```

```python
            print(f"  '{col}': {matches}/{len(expected)} matches ({matches/len(expected)*100:.1f}%)")


def validate_frequency_threshold_strategy(df_original, df_processed, original_column, new_columns, separator):
    """Validate frequency threshold strategy"""

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.split(separator) if v.strip()])

    # Get value counts
    all_values = []
    for values_list in split_original:
        all_values.extend(values_list)
    value_counts = Counter(all_values)

    for col in new_columns:
        if col.endswith('_rare'):
            # Validate rare column - similar to other column validation
            continue
        else:
            # Extract the value name from column name
            value_name = col.replace(f"{original_column}_freq_", "").replace(f"{original_column}_", "")
            validate_binary_column(df_original, df_processed, original_column, col, value_name, separator)


def validate_sample_rows(df_original, df_processed, original_column, new_columns, separator, n_samples=5):
    """Manually validate a few sample rows"""

    print(f"Validating {n_samples} random samples:")

    # Get random sample indices
    sample_indices = np.random.choice(len(df_original), min(n_samples, len(df_original)), replace=False)

    for i, idx in enumerate(sample_indices, 1):
        original_value = df_original.iloc[idx][original_column]
        if pd.isna(original_value):
            original_values = []
        else:
```

```
            original_values = [v.strip() for v in str(original_value).
↪split(separator) if v.strip()]

        print(f"\n  Sample {i} (row {idx}):")
        print(f"    Original: '{original_value}'")
        print(f"    Parsed: {original_values}")

        # Check new columns for this row
        for col in new_columns[:5]:  # Show first 5 columns only
            processed_value = df_processed.iloc[idx][col]
            print(f"    {col}: {processed_value}")


def validate_statistics(df_original, df_processed, original_column,␣
↪new_columns, separator):
    """Validate statistical properties"""

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.
↪split(separator) if v.strip()])

    # Original statistics
    values_per_row = split_original.apply(len)
    print(f"Original values per row - Mean: {values_per_row.mean():.2f}, Std:␣
↪{values_per_row.std():.2f}")

    # New data statistics
    if any('_count' in col for col in new_columns):
        count_col = [col for col in new_columns if col.endswith('_count')][0]
        new_counts = df_processed[count_col]
        print(f"Processed counts - Mean: {new_counts.mean():.2f}, Std:␣
↪{new_counts.std():.2f}")

        # They should match!
        correlation = np.corrcoef(values_per_row, new_counts)[0, 1]
        print(f"Correlation between original and processed counts: {correlation:
↪.4f}")

    # Check for any impossible values
    binary_cols = [col for col in new_columns if not col.endswith(('_count',␣
↪'_frequency', '_avg_frequency'))]
    for col in binary_cols:
        unique_vals = df_processed[col].unique()
        if not set(unique_vals).issubset({0, 1, np.nan}):
            print(f"  Warning: Non-binary values in '{col}': {unique_vals}")
```

```python
def assess_information_loss(df_original, df_processed, original_column, new_columns, separator):
    """Assess how much information was lost in the transformation"""

    # Parse original data
    original_data = df_original[original_column].fillna('').astype(str)
    split_original = original_data.apply(lambda x: [v.strip() for v in x.split(separator) if v.strip()])

    # Get all unique values
    all_original_values = set()
    for values_list in split_original:
        all_original_values.update(values_list)
    all_original_values = sorted([v for v in all_original_values if v and v != 'nan'])

    # Count how many unique values are captured by new columns
    captured_values = set()
    for col in new_columns:
        if not col.endswith(('_other', '_count', '_unique_count', '_frequency', '_avg_frequency', '_rare')):
            # Extract value name from column name
            value_parts = col.replace(f"{original_column}_", "").replace("top_", "").replace("freq_", "")
            captured_values.add(value_parts)

    capture_rate = len(captured_values) / len(all_original_values) * 100 if all_original_values else 0
    print(f"Value capture rate: {len(captured_values)}/{len(all_original_values)} ({capture_rate:.1f}%)")

    if len(all_original_values) - len(captured_values) > 0:
        lost_values = set(all_original_values) - captured_values
        print(f"Lost values (first 10): {list(lost_values)[:10]}")

    # Estimate row-level information preservation
    if any('_other' in col for col in new_columns):
        other_col = [col for col in new_columns if col.endswith('_other')][0]
        rows_with_other = df_processed[other_col].sum()
        print(f"Rows with 'other' values: {rows_with_other}/{len(df_processed)} ({rows_with_other/len(df_processed)*100:.1f}%)")


def quick_validation_summary(df_original, df_processed, column_mapping):
```

```
    """Quick validation summary for all processed columns"""

    print("=== QUICK VALIDATION SUMMARY ===\n")

    for original_col, new_cols in column_mapping.items():
        print(f"  {original_col} → {len(new_cols)} new columns")

        # Check for obvious issues
        issues = []

        for col in new_cols:
            if col not in df_processed.columns:
                issues.append(f"Missing column: {col}")
            else:
                # Check for unexpected values in binary columns
                if not col.endswith(('_count', '_frequency', '_avg_frequency')):
                    unique_vals = set(df_processed[col].dropna().unique())
                    if not unique_vals.issubset({0, 1, 0.0, 1.0}):
                        issues.append(f"Non-binary values in {col}:␣
 ↪{unique_vals}")

        if issues:
            print(f"     Issues: {issues}")
        else:
            print(f"    Looks good")

    print(f"\nDataset size: {len(df_original)} → {len(df_processed)} rows")
    print(f"Column count: {len(df_original.columns)} → {len(df_processed.
 ↪columns)}")
```

[6]:
```
def add_missing_categories_from_full_dataset(
    sample_df,
    full_df,
    categorical_columns,
    samples_per_category=2,
    exclude_columns=None
):
    """
    Add missing categorical values to sample dataset by sampling from full␣
 ↪dataset

    Parameters:
    -----------
    sample_df : pd.DataFrame
        Your limited sample dataset
    full_df : pd.DataFrame
        Your complete dataset
```

```
    categorical_columns : list
        List of categorical column names
    samples_per_category : int
        Number of examples to add for each missing category
    exclude_columns : list
        Columns that should not get ANY new categories (even indirectly)

    Returns:
    --------
    pd.DataFrame : Enhanced dataset with missing categories included
    """

    print("Analyzing missing categories...")

    # Apply exclusions if provided at this level
    if exclude_columns:
        categorical_columns = [col for col in categorical_columns
                              if col not in exclude_columns]
        print(f"Excluded columns: {exclude_columns}")

    # Find missing categories in sample compared to full dataset
    missing_categories = {}
    category_stats = {}

    for col in categorical_columns:
        if col not in sample_df.columns or col not in full_df.columns:
            print(f"Warning: Column '{col}' not found in one of the datasets")
            continue

        full_categories = set(full_df[col].dropna().unique())
        sample_categories = set(sample_df[col].dropna().unique())
        missing = full_categories - sample_categories

        if missing:
            missing_categories[col] = missing
            category_stats[col] = {
                'total_in_full': len(full_categories),
                'in_sample': len(sample_categories),
                'missing_count': len(missing)
            }
            print(f"Column '{col}': Missing {len(missing)} out of␣
↪{len(full_categories)} categories")
            print(f"  Missing categories: {list(missing)[:5]}{'...' if␣
↪len(missing) > 5 else ''}")
        else:
            print(f"Column '{col}': All categories present in sample")
```

```
    if not missing_categories:
        print("No missing categories found! Your sample already contains all␣
↪category values.")
        return sample_df.copy()

    # Collect additional rows for missing categories
    additional_rows = []
    rows_added_by_category = defaultdict(int)

    for col, missing_vals in missing_categories.items():
        print(f"\nSampling for column '{col}'...")

        for val in missing_vals:
            # Find all rows in full dataset with this category value
            matching_rows = full_df[full_df[col] == val]

            if len(matching_rows) == 0:
                print(f"  Warning: No rows found for {col}='{val}' in full␣
↪dataset")
                continue

            # Sample requested number of rows (or all available if fewer)
            n_samples = min(samples_per_category, len(matching_rows))
            sampled_rows = matching_rows.sample(n=n_samples, random_state=42)

            # Filter out rows that would introduce new categories in excluded␣
↪columns
            if exclude_columns:
                original_sample_size = len(sampled_rows)

                for exclude_col in exclude_columns:
                    if exclude_col in sampled_rows.columns and exclude_col in␣
↪sample_df.columns:
                        # Get existing categories in sample
                        existing_categories = set(sample_df[exclude_col].
↪dropna().unique())

                        # Only keep rows where excluded column has existing␣
↪values or is null
                        mask = (sampled_rows[exclude_col].
↪isin(existing_categories) |
                               sampled_rows[exclude_col].isna())
                        sampled_rows = sampled_rows[mask]

                if len(sampled_rows) == 0:
```

```
                    print(f"  Skipped '{val}': Would violate exclusion␣
↪constraints")
                    continue
                elif len(sampled_rows) < original_sample_size:
                    print(f"  Filtered {original_sample_size -␣
↪len(sampled_rows)} rows to respect exclusions")

            additional_rows.append(sampled_rows)
            rows_added_by_category[f"{col}='{val}'"] = len(sampled_rows)
            print(f"  Added {len(sampled_rows)} rows for '{val}' (out of␣
↪{len(matching_rows)} available)")

    # Combine all additional rows
    if additional_rows:
        df_additional = pd.concat(additional_rows, ignore_index=True)

        # Remove potential duplicates (in case same row satisfies multiple␣
↪missing categories)
        initial_additional_count = len(df_additional)
        df_additional = df_additional.drop_duplicates()
        final_additional_count = len(df_additional)

        if initial_additional_count != final_additional_count:
            print(f"\nRemoved {initial_additional_count -␣
↪final_additional_count} duplicate rows")

        # Combine with original sample
        df_enhanced = pd.concat([sample_df, df_additional], ignore_index=True)

        print(f"\n=== SUMMARY ===")
        print(f"Original sample size: {len(sample_df)}")
        print(f"Additional rows added: {len(df_additional)}")
        print(f"Final dataset size: {len(df_enhanced)}")
        print(f"Size increase: {len(df_additional)/len(sample_df)*100:.1f}%")

        return df_enhanced

    else:
        print("No additional rows could be sampled")
        return sample_df.copy()

def verify_categories_coverage(df_before, df_after, categorical_columns):
    """
    Verify that the enhanced dataset now covers all categories
    """
    print("\n=== CATEGORY COVERAGE VERIFICATION ===")
```

```
    for col in categorical_columns:
        if col not in df_before.columns:
            continue

        before_cats = set(df_before[col].dropna().unique())
        after_cats = set(df_after[col].dropna().unique())
        new_cats = after_cats - before_cats

        print(f"\nColumn '{col}':")
        print(f"  Before: {len(before_cats)} categories")
        print(f"  After:  {len(after_cats)} categories")
        if new_cats:
            print(f"  New categories added: {list(new_cats)}")
```

[7]:
```
def create_preserving_pipeline(target_col, max_categorical_cardinality,␣
 ↪excluded_columns=None):
    """
    Create a pipeline that preserves excluded columns
    """
    excluded_columns = excluded_columns or []

    return Pipeline([
        ('multi_value_encoder', PreservingMultiValueEncoder(
            max_cardinality=max_categorical_cardinality,
            preserve_columns=excluded_columns
        )),
        ('categorical_encoder', PreservingCategoricalEncoder(
            max_cardinality=max_categorical_cardinality,
            preserve_columns=excluded_columns
        )),
        ('column_fixer', ColumnNameFixer()),
        ('validator', DataValidator(target_col))
    ])
```

[8]:
```
import pandas as pd
import numpy as np
import re
import os
import joblib
from datetime import datetime
from pathlib import Path
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MultiLabelBinarizer

# === 1. DataLoader: Load data and check target column ===
```

```
class DataLoader(BaseEstimator, TransformerMixin):
    """
        Load and perform initial data validation whether the target col exists:
        - Handles both .xlsx and .csv.
        - Stores the original shape of the data.
        - Raises an error if the target column is missing.

    """

    def __init__(self, file_path,␣
↪target_col='project_prf_normalised_work_effort'):
        self.file_path = file_path
        self.target_col = target_col  # This should be the standardized form
        self.original_shape = None
        self.original_target_col = None  # Store what we actually found

    def fit(self, X=None, y=None):
        return self

    def _standardize_column_name(self, col_name):
        """Convert column name to standardized format"""
        return col_name.strip().lower().replace(' ', '_')

    def _find_target_column(self, df_columns):
        """
        Smart target column finder - handles various formats
        Returns the actual column name from the dataframe
        """
        target_standardized = self.target_col.lower().replace(' ', '_')

        # Try exact match first
        if self.target_col in df_columns:
            return self.target_col

        # Try standardized versions of all columns
        for col in df_columns:
            col_standardized = self._standardize_column_name(col)
            if col_standardized == target_standardized:
                return col

        # If still not found, look for partial matches (for debugging)
        similar_cols = []
        target_words = set(target_standardized.split('_'))
        for col in df_columns:
            col_words = set(self._standardize_column_name(col).split('_'))
            if len(target_words.intersection(col_words)) >= 2:  # At least 2␣
↪words match
```

```
                similar_cols.append(col)

        return None, similar_cols

    def transform(self, X=None):
        """Load data from file with smart column handling"""

        print(f"Loading data from: {self.file_path}")

        # Determine file type and load accordingly; support for Excel or CSV
        if self.file_path.endswith('.xlsx'):
            df = pd.read_excel(self.file_path)
        elif self.file_path.endswith('.csv'):
            df = pd.read_csv(self.file_path)
        else:
            raise ValueError("Unsupported file format. Use .xlsx or .csv")

        self.original_shape = df.shape
        print(f"Loaded data with shape: {df.shape}")

        # Smart target column finding
        result = self._find_target_column(df.columns)

        if isinstance(result, tuple):  # Not found, got similar columns
            actual_col, similar_cols = result
            error_msg = f"Target column '{self.target_col}' not found in data."
            if similar_cols:
                error_msg += f" Similar columns found: {similar_cols}"
            else:
                error_msg += f" Available columns: {list(df.columns)}"
            raise ValueError(error_msg)
        else:
            actual_col = result

        # Store the original column name we found
        self.original_target_col = actual_col

        if actual_col != self.target_col:
            print(f"Target column found: '{actual_col}' -> will be standardized␣
↪to '{self.target_col}'")

        return df


# === 2. ColumnNameStandardizer: Clean and standardize column names ===
class ColumnNameStandardizer(BaseEstimator, TransformerMixin):
    """
```

```
        Standardize column names for consistency (lowercase, underscores,␣
↪removes odd chars):
        - Strips spaces, lowercases, replaces & with _&_, removes special chars.
        - Useful for later steps and compatibility with modeling libraries.)

    """

    def __init__(self, target_col=None, original_target_col=None):
        self.column_mapping = {}
        self.target_col = target_col
        self.original_target_col = original_target_col

    def fit(self, X, y=None):
        return self

    def _standardize_columns(self, columns):
        """Standardize column names"""
        return [col.strip().lower().replace(' ', '_') for col in columns]

    def _clean_column_names(self, columns):
        """Clean column names for compatibility"""
        cleaned_cols = []
        for col in columns:
            # Replace ampersands with _&_ to match expected transformations
            col_clean = col.replace(' & ', '_&_')
            # Remove special characters except underscores and ampersands
            col_clean = re.sub(r'[^\w\s&]', '', col_clean)
            # Replace spaces with underscores
            col_clean = col_clean.replace(' ', '_')
            cleaned_cols.append(col_clean)
        return cleaned_cols

    def transform(self, X):
        """Apply column name standardization"""
        df = X.copy()

        # Store original column names
        original_columns = df.columns.tolist()

        # Apply standardization
        standardized_cols = self._standardize_columns(original_columns)
        cleaned_cols = self._clean_column_names(standardized_cols)

        # Special handling for target column
        if self.original_target_col and self.target_col:
            target_index = None
            try:
```

```
                target_index = original_columns.index(self.original_target_col)
                cleaned_cols[target_index] = self.target_col
                print(f"Target column '{self.original_target_col}' -> '{self.
↪target_col}'")
            except ValueError:
                pass  # Original target col not found, proceed normally


        # Create mapping
        self.column_mapping = dict(zip(original_columns, cleaned_cols))

        # Apply new column names
        df.columns = cleaned_cols

        # Report changes
        changed_cols = sum(1 for orig, new in self.column_mapping.items() if␣
↪orig != new)
        print(f"Standardized {changed_cols} column names")

        return df

# === 3. MissingValueAnalyzer: Analyze and handle missing values ===
class MissingValueAnalyzer(BaseEstimator, TransformerMixin):
    """
        Analyze and handle missing values
        - Reports number of columns with >50% and >70% missing.
        - Drops columns with a high proportion of missing data, except those␣
↪you want to keep.
        - Fills remaining missing values:
            - Categorical: Fills with "Missing".
            - Numeric: Fills with column median.
    """

    def __init__(self, high_missing_threshold=0.7, cols_to_keep=None):
        self.high_missing_threshold = high_missing_threshold
        self.cols_to_keep = cols_to_keep or []
        self.high_missing_cols = []
        self.missing_stats = {}

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        """Analyze and handle missing values"""
        df = X.copy()

        # Calculate missing percentages
```

```python
        missing_pct = df.isnull().mean()
        self.missing_stats = missing_pct.sort_values(ascending=False)

        print(f"\nMissing value analysis:")
        print(f"Columns with >50% missing: {sum(missing_pct > 0.5)}")
        print(f"Columns with >70% missing: {sum(missing_pct > self.high_missing_threshold)}")

        # Identify high missing columns
        self.high_missing_cols = missing_pct[missing_pct > self.high_missing_threshold].index.tolist()

        # Filter out columns we want to keep
        final_high_missing_cols = [col for col in self.high_missing_cols if col not in self.cols_to_keep]

        print(f"Dropping {len(final_high_missing_cols)} columns with >{self.high_missing_threshold*100}% missing values")

        # Drop high missing columns
        df_clean = df.drop(columns=final_high_missing_cols)

        # Fill remaining missing values in categorical columns
        cat_cols = df_clean.select_dtypes(include=['object', 'category']).columns
        for col in cat_cols:
            df_clean[col] = df_clean[col].fillna('Missing')

        # Fill remaining missing values in numerical columns with median
        num_cols = df_clean.select_dtypes(include=['number']).columns
        for col in num_cols:
            if df_clean[col].isnull().sum() > 0:
                median_val = df_clean[col].median()
                df_clean[col] = df_clean[col].fillna(median_val)
                print(f"Filled {col} missing values with median: {median_val}")

        print(f"Data shape after missing value handling: {df_clean.shape}")
        return df_clean

# === 4. SemicolonProcessor: Process multi-value columns (semicolon-separated) ===
class SemicolonProcessor(BaseEstimator, TransformerMixin):
    """
        Process semicolon-separated values in columns (e.g., "Python; Java; SQL")
        - Identifies columns with semicolons.
```

```
        - Cleans: lowercases, strips, deduplicates, sorts, optionally␣
↪standardizes values (e.g., "stand alone" → "stand-alone").
        - Useful for multi-value categorical features.

    """

    def __init__(self, standardization_mapping=None):
        self.semicolon_cols = []
        self.standardization_mapping = standardization_mapping or {
            "scrum": "agile development",
            "file &/or print server": "file/print server",
        }

    def fit(self, X, y=None):
        return self

    def _clean_and_sort_semicolon(self, val, apply_standardization=False,␣
↪mapping=None):
        """Clean, deduplicate, sort, and standardize semicolon-separated␣
↪values"""
        if pd.isnull(val) or val == '':
            return val

        parts = [x.strip().lower() for x in str(val).split(';') if x.strip()]

        if apply_standardization and mapping is not None:
            parts = [mapping.get(part, part) for part in parts]

        unique_cleaned = sorted(set(parts))
        return '; '.join(unique_cleaned)

    def transform(self, X):
        """Process semicolon-separated columns"""
        df = X.copy()

        # Identify columns with semicolons
        self.semicolon_cols = [
            col for col in df.columns
            if df[col].dropna().astype(str).str.contains(';').any()
        ]

        print(f"Found {len(self.semicolon_cols)} columns with semicolons: {self.
↪semicolon_cols}")

        # Process each semicolon column
        for col in self.semicolon_cols:
            # Apply mapping for specific columns
```

```
            apply_mapping = col in ['process_pmf_development_methodologies',␣
↪'tech_tf_server_roles']
            mapping = self.standardization_mapping if apply_mapping else None

            # Clean the column
            df[col] = df[col].apply(
                lambda x: self._clean_and_sort_semicolon(x,␣
↪apply_standardization=apply_mapping, mapping=mapping)
            )

        return df

# === 5. MultiValueEncoder: Encode semicolon columns using MultiLabelBinarizer␣
↪===
class MultiValueEncoder(BaseEstimator, TransformerMixin):
    """
        Handle multi-value columns using MultiLabelBinarizer
        - Only processes columns with a manageable number of unique values␣
↪(max_cardinality).
        - Each semicolon column becomes several binary columns (e.g.,␣
↪"lang__python", "lang__java", ...).
    """

    def __init__(self, max_cardinality=10):
        # Ensure max_cardinality is always an integer
        self.max_cardinality = int(max_cardinality) if max_cardinality is not␣
↪None else 10
        self.multi_value_cols = []
        self.mlb_transformers = {}

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        """Encode multi-value columns"""
        df = X.copy()

        # Identify semicolon columns (multi-value)
        semicolon_cols = [
            col for col in df.columns
            if df[col].dropna().astype(str).str.contains(';').any()
        ]

        # Filter for low cardinality multi-value columns
        self.multi_value_cols = []
        for col in semicolon_cols:
```

```
            # Get unique values across all entries
            all_values = set()
            for val in df[col].dropna().astype(str):
                values = [v.strip() for v in val.split(';') if v.strip()]
                all_values.update(values)

            # Check cardinality (max_cardinality is already an integer from __init__)
            if len(all_values) <= self.max_cardinality:
                self.multi_value_cols.append(col)

        print(f"Encoding {len(self.multi_value_cols)} multi-value columns: {self.multi_value_cols}")

        # Process each multi-value column
        for col in self.multi_value_cols:
            # Prepare data for MultiLabelBinarizer
            values = df[col].dropna().astype(str).apply(
                lambda x: [item.strip() for item in x.split(';') if item.strip()]
            )

            # Handle empty values - fill with empty list for MultiLabelBinarizer
            if len(values) == 0:
                continue

            # Fit and transform
            mlb = MultiLabelBinarizer()

            # Convert to list of lists, handling NaN/empty cases
            values_list = []
            for idx in df.index:
                if idx in values.index and values[idx]:
                    values_list.append(values[idx])
                else:
                    values_list.append([])  # Empty list for missing values

            onehot = pd.DataFrame(
                mlb.fit_transform(values_list),
                columns=[f"{col}__{cat}" for cat in mlb.classes_],
                index=df.index
            )

            # Store transformer for later use
            self.mlb_transformers[col] = mlb

            # Join with main dataframe
```

```
            df = df.join(onehot, how='left')

            print(f"Encoded {col} into {len(mlb.classes_)} binary columns")

        # Remove original multi-value columns
        df = df.drop(columns=self.multi_value_cols)

        return df

class PreservingMultiValueEncoder(MultiValueEncoder):
    """
    Modified MultiValueEncoder that preserves specific columns
    """
    def __init__(self, max_cardinality=10, preserve_columns=None):
        super().__init__(max_cardinality)
        self.preserve_columns = preserve_columns or []

    def transform(self, X):
        """Encode multi-value columns but preserve specified columns"""
        df = X.copy()

        # Store preserved columns before processing
        preserved_data = {}
        for col in self.preserve_columns:
            if col in df.columns:
                preserved_data[col] = df[col].copy()
                print(f"[PreservingMultiValueEncoder] Preserving column␣
↪'{col}'")

        # Apply normal multi-value encoding
        df_processed = super().transform(df)

        # Add back preserved columns (they might have been processed/removed)
        for col, data in preserved_data.items():
            if col not in df_processed.columns:
                df_processed[col] = data
                print(f"[PreservingMultiValueEncoder] Restored column '{col}'")

        return df_processed



# === 6. CategoricalEncoder: One-hot encode regular categorical columns ===
class CategoricalEncoder(BaseEstimator, TransformerMixin):
    """
        Handle single-value categorical columns
        - Ignores semicolon columns.
```

```python
        - Only encodes columns with a number of categories   max_cardinality (to avoid high-dimensional explosion).
        - Can drop the first category for each variable to avoid multicollinearity.

    """

    def __init__(self, max_cardinality=10, drop_first=True):
        self.max_cardinality = max_cardinality
        self.drop_first = drop_first
        self.categorical_cols = []

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        """Encode categorical columns"""
        df = X.copy()

        # Identify categorical columns
        cat_cols = df.select_dtypes(include=['object', 'category']).columns.tolist()

        # Identify semicolon columns to exclude
        semicolon_cols = [
            col for col in df.columns
            if df[col].dropna().astype(str).str.contains(';').any()
        ]

        # Filter for low cardinality single-value categorical columns
        excluded_columns = getattr(self, 'preserve_columns', [])  # Add this attribute
        self.categorical_cols = [
            col for col in cat_cols
            if (col not in semicolon_cols and
                df[col].nunique() <= self.max_cardinality and
                col not in excluded_columns)  # Skip excluded columns
        ]

        print(f"One-hot encoding {len(self.categorical_cols)} categorical columns: {self.categorical_cols}")

        # Apply one-hot encoding
        if self.categorical_cols:
            df = pd.get_dummies(df, columns=self.categorical_cols, drop_first=self.drop_first)
```

```
        return df

# Before creating the final pipeline, modify the encoder classes:
class PreservingCategoricalEncoder(CategoricalEncoder):
    """
    Modified CategoricalEncoder that preserves specific columns
    """
    def __init__(self, max_cardinality=10, drop_first=True,␣
↪preserve_columns=None):
        super().__init__(max_cardinality, drop_first)
        self.preserve_columns = preserve_columns or []

    def transform(self, X):
        """Encode categorical columns but preserve specified columns"""
        df = X.copy()

        # Store preserved columns before processing
        preserved_data = {}
        for col in self.preserve_columns:
            if col in df.columns:
                preserved_data[col] = df[col].copy()
                print(f"[PreservingCategoricalEncoder] Preserving column␣
↪'{col}'")

        # Remove preserved columns from categorical processing
        original_cat_cols = df.select_dtypes(include=['object', 'category']).
↪columns.tolist()
        semicolon_cols = [
            col for col in df.columns
            if df[col].dropna().astype(str).str.contains(';').any()
        ]

        # Filter out preserved columns from processing
        self.categorical_cols = [
            col for col in original_cat_cols
            if (col not in semicolon_cols and
                df[col].nunique() <= self.max_cardinality and
                col not in self.preserve_columns)  # Don't process preserved␣
↪columns
        ]

        print(f"[PreservingCategoricalEncoder] Will encode {len(self.
↪categorical_cols)} columns (excluding {len(self.preserve_columns)}␣
↪preserved)")

        # Apply one-hot encoding to non-preserved columns only
```

```
        if self.categorical_cols:
            df_processed = pd.get_dummies(df, columns=self.categorical_cols,␣
↪drop_first=self.drop_first)
        else:
            df_processed = df.copy()

        # Ensure preserved columns are still there
        for col, data in preserved_data.items():
            if col not in df_processed.columns:
                df_processed[col] = data
                print(f"[PreservingCategoricalEncoder] Restored column '{col}'")

        return df_processed



# === 7. ColumnNameFixer: Final column name cleanup for PyCaret etc ===
class ColumnNameFixer(BaseEstimator, TransformerMixin):
    """
        Fix column names for PyCaret compatibility (removes illegal characters,␣
↪replaces spaces/ampersands, handles duplicates):
        - No duplicate column names after encoding.
        - Only alphanumeric and underscores.

    """

    def __init__(self):
        self.column_transformations = {}

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        """Fix problematic column names"""
        df = X.copy()
        original_cols = df.columns.tolist()
        fixed_columns = []
        seen_columns = set()

        for col in original_cols:
            # Replace spaces with underscores
            fixed_col = col.replace(' ', '_')
            # Replace ampersands
            fixed_col = fixed_col.replace('&', 'and')
            # Remove other problematic characters
            fixed_col = ''.join(c if c.isalnum() or c == '_' else '_' for c in␣
↪fixed_col)
```

```
            # Remove multiple consecutive underscores
            fixed_col = re.sub('_+', '_', fixed_col)
            # Remove leading/trailing underscores
            fixed_col = fixed_col.strip('_')

            # Handle duplicates
            base_col = fixed_col
            suffix = 1
            while fixed_col in seen_columns:
                fixed_col = f"{base_col}_{suffix}"
                suffix += 1

            seen_columns.add(fixed_col)
            fixed_columns.append(fixed_col)

        # Store transformations
        self.column_transformations = dict(zip(original_cols, fixed_columns))

        # Apply new column names
        df.columns = fixed_columns

        # Check for duplicates
        dup_check = [item for item, count in pd.Series(fixed_columns).
↪value_counts().items() if count > 1]
        if dup_check:
            print(f"WARNING: Found {len(dup_check)} duplicate column names:␣
↪{dup_check}")
        else:
            print("No duplicate column names after fixing")

        n_changed = sum(1 for old, new in self.column_transformations.items()␣
↪if old != new)
        print(f"Fixed {n_changed} column names for PyCaret compatibility")

        return df

# === 8. DataValidator: Final summary and checks ===
class DataValidator(BaseEstimator, TransformerMixin):
    """
        Validate final dataset
        - Shape, missing values, infinities.
        - Data types (numeric, categorical).
        - Stats on the target column (mean, std, min, max, missing).
        - Report issues if any.

    """
```

```
    def __init__(self, target_col):
        self.target_col = target_col

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        """Validate the processed dataset"""
        df = X.copy()

        print(f"\n=== Final Data Validation ===")
        print(f"Final shape: {df.shape}")
        print(f"Target column: {self.target_col}")

        # Check for missing values
        missing_count = df.isnull().sum().sum()
        print(f"Total missing values: {missing_count}")

        # Check for infinite values
        numeric_cols = df.select_dtypes(include=[np.number]).columns
        inf_count = np.isinf(df[numeric_cols].values).sum()
        print(f"Total infinite values: {inf_count}")

        # Data types summary
        print(f"\nData types:")
        print(f"  Numeric columns: {len(df.select_dtypes(include=[np.number]).
↪columns)}")
        print(f"  Categorical columns: {len(df.select_dtypes(include=['object',␣
↪'category']).columns)}")

        # Target variable summary
        if self.target_col in df.columns:
            target_stats = df[self.target_col].describe()
            print(f"\nTarget variable '{self.target_col}' statistics:")
            print(f"  Mean: {target_stats['mean']:.2f}")
            print(f"  Std: {target_stats['std']:.2f}")
            print(f"  Min: {target_stats['min']:.2f}")
            print(f"  Max: {target_stats['max']:.2f}")
            print(f"  Missing: {df[self.target_col].isnull().sum()}")
        else:
            print(f"WARNING: Target column '{self.target_col}' not found!")

        return df

# === Pipeline creation function: returns the Scikit-learn pipeline ===
def create_isbsg_preprocessing_pipeline(
    target_col='project_prf_normalised_work_effort',
```

```
    original_target_col=None,
    high_missing_threshold=0.7,
    cols_to_keep=None,
    max_categorical_cardinality=10,
    standardization_mapping=None
):
    """
    Create complete preprocessing pipeline with smart target column handling

    Parameters:
    -----------
    target_col : str
        Name of target column
    original_target_col : str
        Original target column name found in data
    high_missing_threshold : float
        Threshold for dropping columns with high missing values
    cols_to_keep : list
        Columns to keep even if they have high missing values
    max_categorical_cardinality : int
        Maximum number of unique values for categorical encoding
    standardization_mapping : dict
        Custom mapping for standardizing semicolon-separated values

    Returns:
    --------
    sklearn.pipeline.Pipeline
        Complete preprocessing pipeline
    """

    if cols_to_keep is None:
        cols_to_keep = [
            'project_prf_case_tool_used',
            'process_pmf_prototyping_used',
            'tech_tf_client_roles',
            'tech_tf_type_of_server',
            'tech_tf_clientserver_description'
        ]

    # Ensure max_categorical_cardinality is an integer
    if not isinstance(max_categorical_cardinality, int):
        max_categorical_cardinality = 10
        print(f"Warning: max_categorical_cardinality was not an integer,␣
↪defaulting to {max_categorical_cardinality}")

    pipeline = Pipeline([
```

```
        ('column_standardizer', ColumnNameStandardizer(target_col,␣
↪original_target_col)),
        ('missing_handler', MissingValueAnalyzer(
            high_missing_threshold=high_missing_threshold,
            cols_to_keep=cols_to_keep
        )),
        ('semicolon_processor',␣
↪SemicolonProcessor(standardization_mapping=standardization_mapping)),
        ('multi_value_encoder',␣
↪MultiValueEncoder(max_cardinality=max_categorical_cardinality)),
        ('categorical_encoder',␣
↪CategoricalEncoder(max_cardinality=max_categorical_cardinality)),
        ('column_fixer', ColumnNameFixer()),
        ('validator', DataValidator(target_col))
    ])

    return pipeline

# === Full workflow function: orchestrates loading, pipeline, and saving ===
def preprocess_isbsg_data(
    file_path,
    target_col='project_prf_normalised_work_effort',  # Always use standardized␣
↪form
    output_dir='../data',
    save_intermediate=True,
    **pipeline_kwargs
):
    """
    Complete preprocessing workflow for ISBSG data: loads the data, runs
      the full preprocessing pipeline, saves processed data, pipeline
      object, and a metadata report to disk, and returns the processed
      DataFrame and metadata

    Parameters:
    -----------
    file_path : str
        Path to input data file
    target_col : str
        Name of target column
    output_dir : str
        Directory to save processed data
    save_intermediate : bool
        Whether to save intermediate processing steps
    **pipeline_kwargs : dict
        Additional arguments for pipeline creation

    Returns:
```

```
    --------
    pandas.DataFrame
        Processed dataframe ready for modeling
    dict
        Processing metadata and statistics
    """

    # print pipeline header
    print("="*60)
    print("ISBSG Data Preprocessing Pipeline")
    print("="*60)
    print(f"Processing file: {file_path}")
    print(f"Target column (standardized): {target_col}")
    print(f"Timestamp: {datetime.now()}")

    # Create output directory
    os.makedirs(output_dir, exist_ok=True)

    # Load data with smart column detection
    loader = DataLoader(file_path, target_col)
    df_raw = loader.transform(X = None)

    # Create and fit preprocessing pipeline
    pipeline = create_isbsg_preprocessing_pipeline(
        target_col=target_col,
        original_target_col=loader.original_target_col,  # Pass the found␣
↪column name
        **pipeline_kwargs
    )

    # Apply preprocessing in order of ColumnNameStandardizer=>␣
↪MissingValueAnalyzer =>
    # SemicolonProcessor=> MultiValueEncoder=> CategoricalEncoder =>␣
↪ColumnNameFixer

    # Apply preprocessing
    df_processed = pipeline.fit_transform(df_raw)

    # Prepare metadata
    metadata = {
        'original_shape': loader.original_shape,
        'processed_shape': df_processed.shape,
        'processing_timestamp': datetime.now().isoformat(),
        'target_column_standardized': target_col,
        'target_column_original': loader.original_target_col,
        'pipeline_steps': [step[0] for step in pipeline.steps]
    }
```

```
    # Save processed data
    file_stem = Path(file_path).stem
    output_path = os.path.join(output_dir, f"{file_stem}_preprocessed.csv")
    df_processed.to_csv(output_path, index=False)
    print(f"\nProcessed data saved to: {output_path}")

    # Save pipeline
    pipeline_path = os.path.join(output_dir,␣
↪f"{file_stem}_preprocessing_pipeline.pkl")
    joblib.dump(pipeline, pipeline_path)
    print(f"Pipeline saved to: {pipeline_path}")

    # Save metadata
    metadata_path = os.path.join(output_dir,␣
↪f"{file_stem}_preprocessing_metadata.txt")
    with open(metadata_path, 'w') as f:
        f.write("ISBSG Data Preprocessing Metadata\n")
        f.write("="*40 + "\n")
        for key, value in metadata.items():
            f.write(f"{key}: {value}\n")

    print(f"Metadata saved to: {metadata_path}")

    # Print completion & return results
    print("\n" + "="*60)
    print("Preprocessing completed successfully!")
    print("="*60)

    return df_processed, metadata
```

[9]:
```
def integrated_categorical_preprocessing(
    sample_file_path: str,
    full_file_path: str,
    target_col: str,
    output_dir: str,
    cols_to_keep: List[str] = None,
    high_card_columns: List[str] = None,
    process_high_cardinality: bool = True,
    max_categorical_cardinality: int = 10,
    samples_per_category: int = 3,
    standardization_mapping: Dict[str, str] = None,
    high_missing_threshold: float = 0.7,
    separator: str = ';',
    strategy: str = 'top_k',
    k: int = 20,
    exclude_from_enhancement: List[str] = None
```

```
) -> Tuple[pd.DataFrame, Dict[str, Any]]:
    """
    Integrated pipeline to:
    1. Load sample and full datasets
    2. Apply consistent preprocessing to both datasets before comparison
    2. Auto-detect categorical columns
    3. Handle high-cardinality multi-value columns
    4. Enhance sample with missing categories from full dataset
    5. Apply standardization and final preprocessing

    Parameters:
    -----------
    exclude_from_enhancement : List[str]
        List of column names to exclude from getting additional categories from␣
↪full dataset

    Returns:
        - Enhanced and processed DataFrame
        - Metadata about the processing steps
    """

    print("="*60)
    print("INTEGRATED CATEGORICAL PREPROCESSING PIPELINE")
    print("="*60)

    # Initialize exclude list if not provided
    if exclude_from_enhancement is None:
        exclude_from_enhancement = []

    # Step 1: Load datasets
    print("\n1. Loading datasets...")
    sample_df = pd.read_excel(sample_file_path)
    full_df = pd.read_excel(full_file_path)

    # Lowercase all column names in both DataFrames independently
    sample_df.columns = [col1.lower() for col1 in sample_df.columns]
    full_df.columns   = [col2.lower() for col2 in full_df.columns]


    print(f"Sample dataset shape: {sample_df.shape}")
    print(f"Full dataset shape: {full_df.shape}")

    # Step 2: Create preprocessing pipeline (WITHOUT final validation)
    print("\n2. Creating preprocessing pipeline...")

    # Create a pipeline that stops before final validation
    initial_pipeline = Pipeline([
```

```
        ('column_standardizer', ColumnNameStandardizer(target_col, target_col.
↪lower())),
        ('missing_handler', MissingValueAnalyzer(
            high_missing_threshold=high_missing_threshold,
            cols_to_keep=cols_to_keep or []
        )),
        ('semicolon_processor',␣
↪SemicolonProcessor(standardization_mapping=standardization_mapping)),
    ])

    # Step 3: Apply initial preprocessing to BOTH datasets
    print("\n3. Applying initial preprocessing to both datasets...")

    # Process sample dataset
    sample_df_preprocessed = initial_pipeline.fit_transform(sample_df)
    print(f"Sample after initial preprocessing: {sample_df_preprocessed.shape}")

    # Process full dataset with same pipeline
    full_df_preprocessed = initial_pipeline.transform(full_df)  # Use␣
↪transform, not fit_transform
    print(f"Full dataset after initial preprocessing: {full_df_preprocessed.
↪shape}")

    # Step 4: Handle high-cardinality columns on PREPROCESSED datasets
    # Initialize col_mapping regardless of whether we process high-cardinality␣
↪columns
    col_mapping = {}
    if process_high_cardinality and high_card_columns:
        print("\n4. Processing high-cardinality multi-value columns...")

        if high_card_columns is None:
            high_card_columns = ['external_eef_organisation_type',␣
↪'project_prf_application_type']

        # Filter out excluded columns from high-cardinality processing
        high_card_columns_to_process = [col for col in high_card_columns
                                      if col not in (exclude_from_enhancement or [])]

        if not high_card_columns_to_process:
            print("All high-cardinality columns are in exclusion list -␣
↪skipping processing")
        else:
            # Process high-cardinality columns in both datasets
            for col in high_card_columns_to_process:
                if col in full_df_preprocessed.columns:
                    print(f"\nProcessing high-cardinality column: {col}")
```

```
                    # Process full dataset
                    full_df_preprocessed, temp_mapping =␣
↪handle_high_cardinality_multivalue(
                        full_df_preprocessed,
                        multi_value_columns=[col],
                        separator=separator,
                        strategy=strategy,
                        k=k,
                        preserve_original=exclude_from_enhancement
                    )

                    # Process sample dataset with same strategy
                    sample_df_preprocessed, _ =␣
↪handle_high_cardinality_multivalue(
                        sample_df_preprocessed,
                        multi_value_columns=[col],
                        separator=separator,
                        strategy=strategy,
                        k=k,
                        preserve_original=exclude_from_enhancement
                    )

                    col_mapping.update(temp_mapping)
    else:
        print("\n4. Skipping high-cardinality processing (disabled or no␣
↪columns specified)")

    # Step 4.5: Expand exclude list to include all derived binary columns
    print("\n4.5. Expanding exclude list to include derived columns...")
    expanded_exclude_list = exclude_from_enhancement.copy()

    for pattern in exclude_from_enhancement:
        # Find all columns that start with the excluded pattern
        matching_cols = [col for col in sample_df_preprocessed.columns
                        if col.startswith(pattern)]
        expanded_exclude_list.extend(matching_cols)
        if matching_cols:
            print(f"  Added derived columns for '{pattern}': {matching_cols}")

    # Remove duplicates
    expanded_exclude_list = list(set(expanded_exclude_list))
    print(f"  Final expanded exclude list: {len(expanded_exclude_list)}␣
↪columns")

    # Step 5: NOW identify categorical columns from preprocessed datasets
    print("\n5. Identifying categorical columns from preprocessed datasets...")
```

```
    categorical_columns = []
    for col in sample_df_preprocessed.columns:
        if (sample_df_preprocessed[col].dtype == 'object' or
            sample_df_preprocessed[col].nunique() < max_categorical_cardinality):
            categorical_columns.append(col)

    categorical_columns = [col.lower() for col in categorical_columns]
    print(f"Detected categorical columns: {len(categorical_columns)} columns")

    # Step 6: Enhanced category sampling with PROPER exclusions
    print("\n6. Enhancing sample with missing categories from preprocessed full dataset...")
    print(f"Excluding columns from enhancement: {exclude_from_enhancement}")

    # SEPARATE enhancement exclusion from processing exclusion
    columns_to_enhance = [col for col in categorical_columns
                          if col not in (exclude_from_enhancement or [])]

    print(f"Columns that will be enhanced: {len(columns_to_enhance)} out of {len(categorical_columns)}")

    # Add missing categories ONLY for non-excluded columns
    enhanced_df = add_missing_categories_from_full_dataset(
        sample_df=sample_df_preprocessed,
        full_df=full_df_preprocessed,
        categorical_columns=columns_to_enhance,  # Only enhance these
        samples_per_category=samples_per_category,
        exclude_columns=exclude_from_enhancement  # But respect exclusions for side effects
    )

    print(f"Enhanced dataset shape: {enhanced_df.shape}")

    # CRITICAL FIX: Ensure excluded columns are still in enhanced_df
    excluded_columns = exclude_from_enhancement or []
    missing_excluded = []

    for exclude_col in excluded_columns:
        if exclude_col not in enhanced_df.columns:
            missing_excluded.append(exclude_col)
            print(f"   WARNING: Excluded column '{exclude_col}' missing from enhanced_df")

    if missing_excluded:
        print(f"\nERROR: {len(missing_excluded)} excluded columns are missing!")
        print("This should not happen. Checking original preprocessed data...")
```

```
        for col in missing_excluded:
            if col in sample_df_preprocessed.columns:
                print(f"  '{col}' exists in sample_df_preprocessed - copying over")
                enhanced_df[col] = sample_df_preprocessed[col]
            else:
                print(f"  '{col}' not found anywhere!")

    # Step 6.5: Restore excluded columns if they got lost during high-cardinality processing
    print("\n6.5. Restoring excluded columns that were lost during high-cardinality processing...")

    excluded_columns = exclude_from_enhancement or []
    missing_excluded = []

    for exclude_col in excluded_columns:
        if exclude_col not in enhanced_df.columns:
            missing_excluded.append(exclude_col)
            print(f"  WARNING: Excluded column '{exclude_col}' missing from enhanced_df")

    if missing_excluded:
        print(f"\nRestoring {len(missing_excluded)} excluded columns from original data...")

        for col in missing_excluded:
            # Try to find the column in the original sample data before high-cardinality processing
            if col in sample_df.columns:  # Use the very original sample_df
                print(f"  Restoring '{col}' from original sample_df")
                enhanced_df[col] = sample_df[col].iloc[:len(enhanced_df)]  # Match the length
            elif col in sample_df_preprocessed.columns:
                print(f"  Restoring '{col}' from sample_df_preprocessed")
                enhanced_df[col] = sample_df_preprocessed[col].iloc[:len(enhanced_df)]
            else:
                print(f"  ERROR: '{col}' not found in any source data!")

        print(f"Enhanced dataset shape after restoration: {enhanced_df.shape}")
    else:
        print("  All excluded columns are already present")
```

```python
    # Step 7: Verify categories coverage (ALL columns, but note which were excluded)
    print("\n7. Verifying categories coverage...")
    print("\n=== CATEGORY COVERAGE VERIFICATION ===")
    excluded_columns = exclude_from_enhancement or []

    for col in categorical_columns:
        if col not in sample_df_preprocessed.columns:
            continue

        before_cats = set(sample_df_preprocessed[col].dropna().unique())
        after_cats = set(enhanced_df[col].dropna().unique())
        new_cats = after_cats - before_cats

        print(f"\nColumn '{col}':")
        print(f"  Before: {len(before_cats)} categories")
        print(f"  After:  {len(after_cats)} categories")

        if col in excluded_columns:
            if new_cats:
                print(f"    EXCLUDED column gained {len(new_cats)} categories (side effect): {list(new_cats)[:5]}")
            else:
                print(f"   EXCLUDED column preserved (no new categories)")
        else:
            if new_cats:
                print(f"   Enhanced with {len(new_cats)} new categories: {list(new_cats)[:5]}")
            else:
                print(f"   No enhancement needed")

    # Step 8: Apply final preprocessing stages to ALL columns
    print("\n8. Applying final preprocessing stages...")

    # Create modified pipeline that preserves excluded columns
    final_pipeline = create_preserving_pipeline(
        target_col=target_col,
        max_categorical_cardinality=max_categorical_cardinality,
        excluded_columns=exclude_from_enhancement
    )

    final_df = final_pipeline.fit_transform(enhanced_df)

    # Step 9: Final validation and duplicate check
    print("\n9. Final validation and duplicate check...")

    # Check for any remaining duplicates after all processing
```

```python
    final_duplicate_cols = final_df.columns[final_df.columns.duplicated()].
↪tolist()
    if final_duplicate_cols:
        print(f"Warning: Found duplicate columns in final dataset:␣
↪{final_duplicate_cols}")
        final_df = final_df.loc[:, ~final_df.columns.duplicated()]
        print("Removed final duplicate columns")

    print(f"Original sample shape: {sample_df.shape}")
    print(f"Final processed shape: {final_df.shape}")
    print(f"Columns added: {final_df.shape[1] - sample_df.shape[1]}")
    print(f"Rows added: {final_df.shape[0] - sample_df.shape[0]}")

    # Compile metadata
    metadata = {
        'original_sample_shape': sample_df.shape,
        'original_full_shape': full_df.shape,
        'final_shape': final_df.shape,
        'categorical_columns_detected': categorical_columns,
        'high_cardinality_columns_processed': high_card_columns,
        'column_mapping': col_mapping,
        'rows_added_from_full_dataset': final_df.shape[0] - sample_df.shape[0]
    }

    return final_df, metadata

def safe_preprocess_with_fallback(
    enhanced_df: pd.DataFrame,
    target_col: str,
    output_dir: str,
    cols_to_keep: List[str] = None,
    max_categorical_cardinality: int = 10,
    standardization_mapping: Dict[str, str] = None,
    high_missing_threshold: float = 0.7
) -> Tuple[pd.DataFrame, Dict[str, Any]]:
    """
    Safe preprocessing function that handles the file_path requirement
    """

    # Save enhanced dataset to temporary file
    temp_enhanced_path = os.path.join(output_dir, 'temp_enhanced_sample.xlsx')
    enhanced_df.to_excel(temp_enhanced_path, index=False)

    try:
        # Apply preprocessing using existing function
        final_df, preprocessing_metadata = preprocess_isbsg_data(
            file_path=temp_enhanced_path,
```

```
            target_col=target_col,
            output_dir=output_dir,
            cols_to_keep=cols_to_keep,
            max_categorical_cardinality=max_categorical_cardinality,
            standardization_mapping=standardization_mapping,
            high_missing_threshold=high_missing_threshold
        )

        return final_df, preprocessing_metadata

    finally:
        # Clean up temporary file
        try:
            os.remove(temp_enhanced_path)
        except:
            print(f"Warning: Could not remove temporary file␣
 ↪{temp_enhanced_path}")

    return enhanced_df, {'error': 'Preprocessing failed'}
```

[10]:
```
import os

# Configuration constants (define these at module level)
#DATA_FOLDER = "../data"  # Update this path as needed
#SAMPLE_FILE = "sample_data.xlsx"  # Update this filename as needed
#FULL_FILE = "full_data.xlsx"  # Update this filename as needed
#TARGET_COL = "project_prf_normalised_work_effort"

print(f"\nDATA_FOLDER = {DATA_FOLDER}, SAMPLE_FILE = {SAMPLE_FILE}, FULL_FILE =␣
 ↪{FULL_FILE}, TARGET_COL = {TARGET_COL}")

# Main execution function
def main():
    """
    Main function to run the integrated pipeline
    """

    # Configuration
    sample_file_path = os.path.join(CONFIG_FOLDER, SAMPLE_FILE)
    full_file_path = os.path.join(DATA_FOLDER, FULL_FILE)
    FINANCE = "finance"


        # Columns to exclude (customize as needed)
    cols_to_exclude_add_category = [
        'external_eef_industry_sector',
        'external_eef_organisation_type',
```

```
        'project_prf_application_type',
     ]

    # Columns to keep (customize as needed)
    cols_to_keep = [
        'Project_PRF_CASE_Tool_Used',
        'Process_PMF_Prototyping_Used',
        'Tech_TF_Client_Roles',
        'Tech_TF_Type_of_Server',
        'Tech_TF_ClientServer_Description'
    ]

    # High-cardinality multi-value columns: will top_k strategy - Keep only top␣
↪K most frequent categorical values)
    high_card_columns = [
        'external_eef_organisation_type',
        'project_prf_application_type'
    ]

    # Standardization rules
    standardization_map = {
        'stand alone': 'stand-alone',
        'client server': 'client-server',
        'mathematically intensive': 'mathematically-intensive',
        #'mathematically intensive application': 'mathematically-intensive␣
↪application',
        "file &/or print server": "file/print server",
    }

    try:
        # Run integrated pipeline
        final_df, metadata = integrated_categorical_preprocessing(
            sample_file_path=sample_file_path,
            full_file_path=full_file_path,
            target_col=TARGET_COL,
            output_dir=DATA_FOLDER,
            cols_to_keep=cols_to_keep,
            high_card_columns=high_card_columns,
            max_categorical_cardinality=10,
            samples_per_category=3,
            standardization_mapping=standardization_map,
            high_missing_threshold=0.7,
            separator=';',
            strategy='top_k',
            k=20,
            process_high_cardinality=False,
            exclude_from_enhancement=cols_to_exclude_add_category
```

```
        )

        # Save results
        if FINANCE in sample_file_path:
            output_path = os.path.join(DATA_FOLDER,␣
↪f"{FINANCE}_enhanced_sample_final.csv")
        else:
            output_path = os.path.join(DATA_FOLDER, 'enhanced_sample_final.csv')

        final_df.to_csv(output_path, index=False)

        # Check what columns are actually in final_df
        print(f"\\n=== FINAL COLUMN CHECK ===")
        print(f"Total columns in final CSV: {len(final_df.columns)}")
        print(f"All columns: {list(final_df.columns)}")


        print(f"\n" + "="*60)
        print("PIPELINE COMPLETED SUCCESSFULLY!")
        print("="*60)
        print(f"Final dataset saved to: {output_path}")
        print(f"Final shape: {final_df.shape}")
        print(f"Ready for PyCaret setup!")

        # Print summary of changes
        print(f"\nSUMMARY:")
        print(f"- Original sample rows: {metadata['original_sample_shape'][0]}")
        print(f"- Rows added from full dataset:␣
↪{metadata['rows_added_from_full_dataset']}")
        print(f"- Final rows: {metadata['final_shape'][0]}")
        print(f"- Original columns: {metadata['original_sample_shape'][1]}")
        print(f"- Final columns: {metadata['final_shape'][1]}")

        return final_df, metadata

    except Exception as e:
        print(f"Error in integrated pipeline: {e}")
        raise
```

DATA_FOLDER = ../data, SAMPLE_FILE = ISBSG2016R1_1_financial_industry_seed.xlsx, FULL_FILE = ISBSG2016R1_1_full_dataset.xlsx, TARGET_COL = project_prf_normalised_work_effort

[ ]:

```

```

[11]:
```python
# Run the main function when script is executed directly
if __name__ == "__main__":
    final_df, metadata = main()
```

```
============================================================
INTEGRATED CATEGORICAL PREPROCESSING PIPELINE
============================================================

1. Loading datasets…
Sample dataset shape: (939, 51)
Full dataset shape: (7518, 52)

2. Creating preprocessing pipeline…

3. Applying initial preprocessing to both datasets…
Standardized 26 column names

Missing value analysis:
Columns with >50% missing: 29
Columns with >70% missing: 26
Dropping 26 columns with >70.0% missing values
Filled project_prf_functional_size missing values with median: 154.5
Filled project_prf_normalised_work_effort_level_1 missing values with median:
1550.0
Filled project_prf_normalised_work_effort missing values with median: 1652.0
Filled project_prf_normalised_level_1_pdr_ufp missing values with median: 11.45
Filled project_prf_normalised_pdr_ufp missing values with median: 11.9
Filled project_prf_speed_of_delivery missing values with median: 27.05
Filled project_prf_project_elapsed_time missing values with median: 6.0
Data shape after missing value handling: (939, 25)
Found 2 columns with semicolons: ['external_eef_organisation_type',
'project_prf_application_type']
Sample after initial preprocessing: (939, 25)
Standardized 27 column names

Missing value analysis:
Columns with >50% missing: 30
Columns with >70% missing: 24
Dropping 24 columns with >70.0% missing values
Filled project_prf_functional_size missing values with median: 139.0
Filled project_prf_normalised_work_effort_level_1 missing values with median:
1593.0
Filled project_prf_normalised_work_effort missing values with median: 1699.0
Filled project_prf_normalised_level_1_pdr_ufp missing values with median: 11.2
Filled project_prf_normalised_pdr_ufp missing values with median: 11.6
Filled project_prf_speed_of_delivery missing values with median: 26.8
Filled project_prf_project_elapsed_time missing values with median: 6.0
Filled project_prf_max_team_size missing values with median: 7.0
```

```
Data shape after missing value handling: (7518, 28)
Found 4 columns with semicolons: ['external_eef_organisation_type',
'project_prf_application_group', 'project_prf_application_type',
'process_pmf_development_methodologies']
Full dataset after initial preprocessing: (7518, 28)

4. Skipping high-cardinality processing (disabled or no columns specified)

4.5. Expanding exclude list to include derived columns…
  Added derived columns for 'external_eef_industry_sector':
['external_eef_industry_sector']
  Added derived columns for 'external_eef_organisation_type':
['external_eef_organisation_type']
  Added derived columns for 'project_prf_application_type':
['project_prf_application_type']
  Final expanded exclude list: 3 columns

5. Identifying categorical columns from preprocessed datasets…
Detected categorical columns: 14 columns

6. Enhancing sample with missing categories from preprocessed full dataset…
Excluding columns from enhancement: ['external_eef_industry_sector',
'external_eef_organisation_type', 'project_prf_application_type']
Columns that will be enhanced: 11 out of 14
Analyzing missing categories…
Excluded columns: ['external_eef_industry_sector',
'external_eef_organisation_type', 'project_prf_application_type']
Column 'external_eef_data_quality_rating': All categories present in sample
Column 'project_prf_application_group': Missing 7 out of 7 categories
  Missing categories: ['real-time application', 'infrastructure software',
'mathematically-intensive application', 'mathematically intensive application',
'business application']…
Column 'project_prf_development_type': Missing 2 out of 7 categories
  Missing categories: ['Porting', 'Other']
Column 'tech_tf_development_platform': Missing 1 out of 7 categories
  Missing categories: ['Hand Held']
Column 'tech_tf_language_type': Missing 1 out of 7 categories
  Missing categories: ['APG']
Column 'tech_tf_primary_programming_language': Missing 80 out of 129 categories
  Missing categories: ['ColdFusion', 'BPM', 'Azure', 'gcc', 'Adobe Flex']…
Column 'project_prf_relative_size': Missing 1 out of 10 categories
  Missing categories: ['XXXL']
Column 'project_prf_case_tool_used': All categories present in sample
Column 'tech_tf_architecture': Missing 2 out of 8 categories
  Missing categories: ['Multi-tier with web interface', 'Stand-alone']
Column 'tech_tf_client_server': All categories present in sample
Column 'tech_tf_dbms_used': All categories present in sample
```

```
Sampling for column 'project_prf_application_group'…
  Skipped 'real-time application': Would violate exclusion constraints
  Skipped 'infrastructure software': Would violate exclusion constraints
  Skipped 'mathematically-intensive application': Would violate exclusion
constraints
  Skipped 'mathematically intensive application': Would violate exclusion
constraints
  Filtered 2 rows to respect exclusions
  Added 1 rows for 'business application' (out of 4655 available)
  Filtered 2 rows to respect exclusions
  Added 1 rows for 'missing' (out of 2311 available)
  Skipped 'business application; infrastructure software': Would violate
exclusion constraints

Sampling for column 'project_prf_development_type'…
  Skipped 'Porting': Would violate exclusion constraints
  Skipped 'Other': Would violate exclusion constraints

Sampling for column 'tech_tf_development_platform'…
  Skipped 'Hand Held': Would violate exclusion constraints

Sampling for column 'tech_tf_language_type'…
  Skipped 'APG': Would violate exclusion constraints

Sampling for column 'tech_tf_primary_programming_language'…
  Skipped 'ColdFusion': Would violate exclusion constraints
  Skipped 'BPM': Would violate exclusion constraints
  Skipped 'Azure': Would violate exclusion constraints
  Skipped 'gcc': Would violate exclusion constraints
  Skipped 'Adobe Flex': Would violate exclusion constraints
  Skipped 'Pega Workflows': Would violate exclusion constraints
  Skipped 'STAFFWARE': Would violate exclusion constraints
  Skipped 'IEF': Would violate exclusion constraints
  Skipped 'Huron/Object Star': Would violate exclusion constraints
  Skipped 'OutlookVBA': Would violate exclusion constraints
  Skipped 'VisualFoxPro': Would violate exclusion constraints
  Skipped 'PYTHON': Would violate exclusion constraints
  Skipped 'MS-Navision Properitory Language': Would violate exclusion
constraints
  Skipped 'Siebel': Would violate exclusion constraints
  Skipped 'ADO.Net': Would violate exclusion constraints
  Skipped 'Delphi': Would violate exclusion constraints
  Skipped 'A:G': Would violate exclusion constraints
  Skipped 'RPL': Would violate exclusion constraints
  Skipped 'COGNOS': Would violate exclusion constraints
  Skipped 'BASIC': Would violate exclusion constraints
  Skipped 'APPS': Would violate exclusion constraints
  Skipped 'INGRES': Would violate exclusion constraints
```

```
  Skipped 'BEA Weblogic': Would violate exclusion constraints
  Skipped 'FORTRAN': Would violate exclusion constraints
  Skipped 'ARBOR/BP': Would violate exclusion constraints
  Skipped 'UNIFACE': Would violate exclusion constraints
  Skipped 'Data base language': Would violate exclusion constraints
  Skipped 'Periproducer': Would violate exclusion constraints
  Skipped 'COOL:Gen': Would violate exclusion constraints
  Skipped 'Centura': Would violate exclusion constraints
  Skipped 'Datastage': Would violate exclusion constraints
  Skipped 'Ada': Would violate exclusion constraints
  Skipped 'Upfront': Would violate exclusion constraints
  Skipped 'SAS': Would violate exclusion constraints
  Skipped 'Doc1 Designer (Entorno visual)': Would violate exclusion constraints
  Skipped 'Informatica PowerCenter': Would violate exclusion constraints
  Skipped 'ACCEL': Would violate exclusion constraints
  Skipped 'Visual Studio .Net': Would violate exclusion constraints
  Skipped 'SLOGAN': Would violate exclusion constraints
  Skipped 'Enablon': Would violate exclusion constraints
  Skipped 'Caa': Would violate exclusion constraints
  Skipped 'HPS': Would violate exclusion constraints
  Skipped 'J2EE': Would violate exclusion constraints
  Skipped 'MATLAB': Would violate exclusion constraints
  Skipped 'LISP': Would violate exclusion constraints
  Skipped 'BRE': Would violate exclusion constraints
  Skipped 'XGML': Would violate exclusion constraints
  Skipped 'BO': Would violate exclusion constraints
  Skipped 'Express': Would violate exclusion constraints
  Skipped 'AB INITIO': Would violate exclusion constraints
  Skipped 'iPlanet Netscape Application Server': Would violate exclusion
constraints
  Skipped 'PERIPHONICS': Would violate exclusion constraints
  Skipped 'EJB': Would violate exclusion constraints
  Skipped 'PHP': Would violate exclusion constraints
  Skipped 'TNSDL': Would violate exclusion constraints
  Skipped 'IBM WTX': Would violate exclusion constraints
  Skipped 'ADS/Online': Would violate exclusion constraints
  Skipped 'Magic': Would violate exclusion constraints
  Skipped 'Object oriented language': Would violate exclusion constraints
  Skipped 'REXX': Would violate exclusion constraints
  Skipped 'ABF': Would violate exclusion constraints
  Skipped 'MANTIS': Would violate exclusion constraints
  Skipped 'ASAP': Would violate exclusion constraints
  Skipped 'NCR teradata scripting': Would violate exclusion constraints
  Skipped 'Jdeveloper': Would violate exclusion constraints
  Skipped 'C/AL': Would violate exclusion constraints
  Skipped 'IIS': Would violate exclusion constraints
  Skipped 'LEX': Would violate exclusion constraints
  Skipped 'Perl': Would violate exclusion constraints
```

```
  Skipped 'IDEAL': Would violate exclusion constraints
  Skipped 'Must Modeller': Would violate exclusion constraints
  Skipped 'Brightware proprietary': Would violate exclusion constraints
  Skipped 'SLEL': Would violate exclusion constraints
  Skipped 'PASCAL': Would violate exclusion constraints
  Skipped 'Spreadsheet': Would violate exclusion constraints
  Skipped 'Formspath': Would violate exclusion constraints
  Skipped 'DRIFT': Would violate exclusion constraints
  Skipped 'PowerPlay': Would violate exclusion constraints
  Skipped 'Mendix': Would violate exclusion constraints
  Skipped 'CICS': Would violate exclusion constraints

Sampling for column 'project_prf_relative_size'…
  Skipped 'XXXL': Would violate exclusion constraints

Sampling for column 'tech_tf_architecture'…
  Skipped 'Multi-tier with web interface': Would violate exclusion constraints
  Skipped 'Stand-alone': Would violate exclusion constraints

=== SUMMARY ===
Original sample size: 939
Additional rows added: 2
Final dataset size: 941
Size increase: 0.2%
Enhanced dataset shape: (941, 28)

6.5. Restoring excluded columns that were lost during high-cardinality
processing…
 All excluded columns are already present

7. Verifying categories coverage…

=== CATEGORY COVERAGE VERIFICATION ===

Column 'external_eef_data_quality_rating':
  Before: 4 categories
  After:  4 categories
   No enhancement needed

Column 'external_eef_industry_sector':
  Before: 2 categories
  After:  2 categories
   EXCLUDED column preserved (no new categories)

Column 'external_eef_organisation_type':
  Before: 16 categories
  After:  16 categories
   EXCLUDED column preserved (no new categories)
```

```
Column 'project_prf_application_group':
  Before: 5 categories
  After:  7 categories
   Enhanced with 2 new categories: ['missing', 'business application']

Column 'project_prf_application_type':
  Before: 51 categories
  After:  51 categories
   EXCLUDED column preserved (no new categories)

Column 'project_prf_development_type':
  Before: 5 categories
  After:  5 categories
   No enhancement needed

Column 'tech_tf_development_platform':
  Before: 6 categories
  After:  6 categories
   No enhancement needed

Column 'tech_tf_language_type':
  Before: 6 categories
  After:  6 categories
   No enhancement needed

Column 'tech_tf_primary_programming_language':
  Before: 49 categories
  After:  49 categories
   No enhancement needed

Column 'project_prf_relative_size':
  Before: 9 categories
  After:  9 categories
   No enhancement needed

Column 'project_prf_case_tool_used':
  Before: 4 categories
  After:  4 categories
   No enhancement needed

Column 'tech_tf_architecture':
  Before: 6 categories
  After:  6 categories
   No enhancement needed

Column 'tech_tf_client_server':
  Before: 5 categories
```

```
  After:  5 categories
    No enhancement needed

Column 'tech_tf_dbms_used':
  Before: 3 categories
  After:  3 categories
    No enhancement needed

8. Applying final preprocessing stages…
[PreservingMultiValueEncoder] Preserving column 'external_eef_industry_sector'
[PreservingMultiValueEncoder] Preserving column 'external_eef_organisation_type'
[PreservingMultiValueEncoder] Preserving column 'project_prf_application_type'
Encoding 0 multi-value columns: []
[PreservingCategoricalEncoder] Preserving column 'external_eef_industry_sector'
[PreservingCategoricalEncoder] Preserving column
'external_eef_organisation_type'
[PreservingCategoricalEncoder] Preserving column 'project_prf_application_type'
[PreservingCategoricalEncoder] Will encode 12 columns (excluding 3 preserved)
No duplicate column names after fixing
Fixed 12 column names for PyCaret compatibility

=== Final Data Validation ===
Final shape: (941, 62)
Target column: project_prf_normalised_work_effort
Total missing values: 939
Total infinite values: 0

Data types:
  Numeric columns: 12
  Categorical columns: 4

Target variable 'project_prf_normalised_work_effort' statistics:
  Mean: 4139.71
  Std: 9330.86
  Min: 6.00
  Max: 134211.00
  Missing: 0

9. Final validation and duplicate check…
Original sample shape: (939, 51)
Final processed shape: (941, 62)
Columns added: 11
Rows added: 2
\n=== FINAL COLUMN CHECK ===
Total columns in final CSV: 62
All columns: ['isbsg_project_id', 'project_prf_year_of_project',
'external_eef_industry_sector', 'external_eef_organisation_type',
'project_prf_application_type', 'tech_tf_primary_programming_language',
```

```
'project_prf_functional_size', 'project_prf_normalised_work_effort_level_1',
'project_prf_normalised_work_effort', 'project_prf_normalised_level_1_pdr_ufp',
'project_prf_normalised_pdr_ufp', 'project_prf_speed_of_delivery',
'project_prf_project_elapsed_time', 'process_pmf_docs', 'tech_tf_tools_used',
'project_prf_max_team_size', 'external_eef_data_quality_rating_B',
'external_eef_data_quality_rating_C', 'external_eef_data_quality_rating_D',
'project_prf_application_group_Infrastructure_Software',
'project_prf_application_group_Mathematically_Intensive_Application',
'project_prf_application_group_Missing',
'project_prf_application_group_Real_Time_Application',
'project_prf_application_group_business_application',
'project_prf_application_group_missing',
'project_prf_development_type_New_Development',
'project_prf_development_type_Not_Defined', 'project_prf_development_type_POC',
'project_prf_development_type_Re_development',
'tech_tf_development_platform_MR', 'tech_tf_development_platform_Missing',
'tech_tf_development_platform_Multi', 'tech_tf_development_platform_PC',
'tech_tf_development_platform_Proprietary', 'tech_tf_language_type_3GL',
'tech_tf_language_type_4GL', 'tech_tf_language_type_5GL',
'tech_tf_language_type_ApG', 'tech_tf_language_type_Missing',
'project_prf_relative_size_M1', 'project_prf_relative_size_M2',
'project_prf_relative_size_Missing', 'project_prf_relative_size_S',
'project_prf_relative_size_XL', 'project_prf_relative_size_XS',
'project_prf_relative_size_XXL', 'project_prf_relative_size_XXS',
'project_prf_case_tool_used_Missing', 'project_prf_case_tool_used_No',
'project_prf_case_tool_used_Yes', 'tech_tf_architecture_Missing',
'tech_tf_architecture_Multi_tier',
'tech_tf_architecture_Multi_tier_Client_server',
'tech_tf_architecture_Multi_tier_with_web_public_interface',
'tech_tf_architecture_Stand_alone', 'tech_tf_client_server_Missing',
'tech_tf_client_server_No', 'tech_tf_client_server_Not_Applicable',
'tech_tf_client_server_Yes', 'tech_tf_dbms_used_No', 'tech_tf_dbms_used_Yes',
'project_prf_team_size_group_Missing']

============================================================
PIPELINE COMPLETED SUCCESSFULLY!
============================================================
Final dataset saved to: ../data\enhanced_sample_final.csv
Final shape: (941, 62)
Ready for PyCaret setup!

SUMMARY:
- Original sample rows: 939
- Rows added from full dataset: 2
- Final rows: 941
- Original columns: 51
- Final columns: 62
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: