THE TECHNICAL MEMOIR

A Cell-by-Cell Journey through the CLV Pipeline

Introduction: The Genesis

The dataset represents 9,134 lives. Every row is a policyholder with a story. In this document, we peel back the layers of abstraction to show you the code that reveals these stories. This is not a summary; it is the raw log of our discovery.

Variable: Customer (object)

This vector controls a specific dimension of risk. Unique cardinality: 9134.

Variable: State (object)

This vector controls a specific dimension of risk. Unique cardinality: 5.

Variable: Customer Lifetime Value (float64)

This vector controls a specific dimension of risk. Unique cardinality: 8041.

Variable: Response (object)

This vector controls a specific dimension of risk. Unique cardinality: 2.

Variable: Coverage (object)

This vector controls a specific dimension of risk. Unique cardinality: 3.

Variable: Education (object)

This vector controls a specific dimension of risk. Unique cardinality: 5.

Variable: Effective To Date (object)

This vector controls a specific dimension of risk. Unique cardinality: 59.

Variable: EmploymentStatus (object)

This vector controls a specific dimension of risk. Unique cardinality: 5.

Variable: Gender (object)

This vector controls a specific dimension of risk. Unique cardinality: 2.

Variable: Income (int64)

This vector controls a specific dimension of risk. Unique cardinality: 5694.

Chapter 1: The Forensic Audit

Notebook 01: Data Loading, Validation, and Cleaning --- Executive Summary This notebook serves as the foundational step in our Customer Lifetime Value (CLV) prediction pipeline. Before any sophisticated modeling can take place, we must ensure our data is **clean, consistent, and well-understood**. Think of this as the "pre-flight checklist" before launching a rocket—every sensor must be verified, every system calibrated. What This Notebook Covers: 1. **Project Context & Business Problem** — Understanding *why* we're doing this analysis 2. **Data Ingestion** — Loading the raw dataset with proper validation 3. **Schema Inspection** — Understanding data types, column semantics, and structure 4. **Data Quality Assessment** — Detecting missing values, duplicates, and anomalies 5. **Data Cleaning Pipeline** — Standardizing strings, fixing dates, and preparing data for analysis 6. **Quality Assurance Checkpoint** — Saving a verified, clean dataset for downstream use --- 1. Project Context and Business Problem 1.1 The Business Challenge In the fiercely competitive automobile insurance industry, understanding the **long-term value of each customer** is not just useful—it's essential for survival. Companies that can accurately predict which customers will generate the most revenue over their lifetime can: - **Optimize Acquisition Budgets**: Spend more to acquire high-value customers, less on low-value ones - **Improve Retention Strategies**: Focus retention efforts on customers who are worth keeping - **Personalize Product Offerings**: Tailor coverage packages to specific value segments - **Forecast Revenue**: Accurately project future income based on current customer base 1.2 The Dataset We are working with the **Watson Analytics Marketing Customer Value Analysis** dataset, which contains detailed records of automobile insurance customers. This dataset is particularly valuable because it includes: | Category | Information Available | |---|---| | **Demographics** | State, Gender, Education, Marital Status, Income | | **Policy Details** | Coverage Level, Policy Type, Monthly Premium, Renewal Offers | | **Behavioral Data** | Number of Policies, Open Complaints, Claims History | | **Target Variable** | Customer Lifetime Value (CLV) in dollars | 1.3 Research Questions Before diving into the data, let's articulate the questions we aim to answer: 1. **Primary Question**: Can we accurately predict Customer Lifetime Value using demographic and policy information? 2. **Secondary Questions**: - Which features are the strongest predictors of CLV? - Are there distinct customer segments with different value profiles? - What actionable insights can we derive for marketing strategy? ---

2. Environment Setup and Library Imports We begin by importing all necessary libraries. This section serves as a **manifest of our technical dependencies**—critical for reproducibility. Why These Libraries? | Library | Purpose | Version Used | |---|---|---| | `pandas` | Data manipulation and analysis | >= 1.5.0 | | `numpy` | Numerical computing | >= 1.21.0 | | `matplotlib` | Static visualizations | >= 3.5.0 | | `seaborn` | Statistical visualizations | >= 0.12.0 | | `os` | File system operations | Built-in |

EXECUTION CODE:

```
# ============================================================
# ENVIRONMENT SETUP
# ============================================================

# Core Data Libraries
import pandas as pd
import numpy as np

# Visualization Libraries
import matplotlib.pyplot as plt
import seaborn as sns

# System Libraries
import os
import warnings
from datetime import datetime

# Display Settings
pd.set_option('display.max_columns', None)  # Show all columns
pd.set_option('display.max_rows', 100)       # Limit row display
pd.set_option('display.float_format', '{:.2f}'.format)  # Format f

# Visualization Settings
plt.style.use('seaborn-v0_8-whitegrid')  # Clean, professional sty
sns.set_palette('viridis')                # Colorblind-friendly pa
plt.rcParams['figure.figsize'] = (12, 6)  # Default figure size
plt.rcParams['font.size'] = 11            # Readable font size

# Suppress Warnings (for clean output)
warnings.filterwarnings('ignore')

# Print Environment Info
print("=" * 60)
print("ENVIRONMENT CONFIGURATION")
print("=" * 60)
print(f"Pandas Version:     {pd.__version__}")
print(f"NumPy Version:      {np.__version__}")
print(f"Execution Time:     {datetime.now().strftime('%Y-%m-%d %H:
print("=" * 60)
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

3. Configuration and Path Management Centralizing all file paths and configuration parameters in one place is a **best practice** that makes the code: - Easier to maintain (change once, apply everywhere) - More portable (adapt paths for different environments) - Self-documenting (all settings visible in one location)

EXECUTION CODE:

```
# ============================================================
# CONFIGURATION
# ============================================================

# Define Base Paths
BASE_DIR = os.path.dirname(os.getcwd())  # Parent of notebooks fol
DATA_RAW_DIR = os.path.join(BASE_DIR, 'data', 'raw')
DATA_PROCESSED_DIR = os.path.join(BASE_DIR, 'data', 'processed')
FIGURES_DIR = os.path.join(BASE_DIR, 'report', 'figures')

# Source Data File
DATA_FILE = 'WA_Fn-UseC_-Marketing-Customer-Value-Analysis.csv'
DATA_PATH = os.path.join(DATA_RAW_DIR, DATA_FILE)
```

```
# Output Files
CLEAN_DATA_PATH = os.path.join(DATA_PROCESSED_DIR, 'cleaned_data.csv')
BACKUP_PATH = os.path.join(DATA_PROCESSED_DIR, 'raw_backup.csv')

# Create directories if they don't exist
for directory in [DATA_RAW_DIR, DATA_PROCESSED_DIR, FIGURES_DIR]:
    os.makedirs(directory, exist_ok=True)

# Verify Configuration
print("=" * 60)
print("PATH CONFIGURATION")
print("=" * 60)
print(f"Base Directory:        {BASE_DIR}")
print(f"Raw Data Directory:    {DATA_RAW_DIR}")
print(f"Processed Data Dir:    {DATA_PROCESSED_DIR}")
print(f"Figures Directory:     {FIGURES_DIR}")
print(f"Source Data File:      {DATA_PATH}")
print(f"\nFile Exists: {os.path.exists(DATA_PATH)}")
print("=" * 60)
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 4. Data Loading with Validation Loading data might seem trivial, but proper validation at this stage prevents countless headaches later. We implement a **defensive loading strategy** that: 1. Verifies file existence before attempting to read 2. Reports file metadata (size, modification date) 3. Captures the raw state before any transformations 4. Provides immediate feedback on data dimensions Why This Matters in Practice In production environments, data files can: - Be corrupted during transfer - Have schema changes from upstream systems - Arrive empty or partially written Validating early catches these issues before they cascade into mysterious errors downstream.

EXECUTION CODE:

```
# ============================================================================
# DATA LOADING WITH VALIDATION
# ============================================================================

def load_data_with_validation(filepath):
    """
    Load CSV data with comprehensive validation.

    Parameters:
    -----------
    filepath : str
        Path to the CSV file

    Returns:
    --------
    pd.DataFrame
        Loaded dataframe

    Raises:
    -------
    FileNotFoundError
        If the file doesn't exist
    ValueError
        If the file is empty
    """

    # Step 1: Verify file exists
    if not os.path.exists(filepath):
        raise FileNotFoundError(f"Data file not found: {filepath}")

    # Step 2: Get file metadata
    file_size = os.path.getsize(filepath)
    file_modified = datetime.fromtimestamp(os.path.getmtime(filepath))

    print(f"■ Loading: {os.path.basename(filepath)}")
    print(f"   Size: {file_size / 1024:.2f} KB ({file_size / (1024*1024):.2f} MB)")
    print(f"   Last Modified: {file_modified.strftime('%Y-%m-%d %H:%M:%S')}")

    # Step 3: Load the data
    df = pd.read_csv(filepath)

    # Step 4: Basic validation
    if df.empty:
        raise ValueError("Loaded DataFrame is empty!")

    # Step 5: Report dimensions
    print(f"\n■ Successfully loaded {len(df):,} rows × {len(df.columns)} columns")

    return df

# Load the data
df_raw = load_data_with_validation(DATA_PATH)
```

OBJECTIVE: We initiate the pipeline by ingesting the raw CSV. This action loads the data into memory, allowing us to perform the initial schema validation. Assessing the magnitude of the dataframe is critical at this stage.

4.1 First Look at the Raw Data The `.head()` method shows us the first few rows. This is our **initial reconnaissance**—what does the data actually look like? Pay attention to: - Column naming conventions (spacing, capitalization) - Apparent data types (numbers, text, dates) - Any obvious formatting issues

EXECUTION CODE:

```
# Display first 5 rows
print("=" * 80)
print("FIRST 5 ROWS OF RAW DATA")
print("=" * 80)
df_raw.head()
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Display last 5 rows (to check for truncation issues)
print("=" * 80)
print("LAST 5 ROWS OF RAW DATA (Checking for truncation)")
print("=" * 80)
df_raw.tail()
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 5. Schema Inspection and Data Profiling Understanding the **schema** (structure) of our data is critical. We need to know: 1. **What columns exist?** — The feature set available to us 2. **What are their data types?** — Determines valid operations 3. **Are there missing values?** — Impacts analysis strategies 4. **What are the value ranges?** — Helps identify anomalies 5.1 DataFrame Info (Technical Schema)

EXECUTION CODE:

```
# Comprehensive schema information
print("=" * 80)
print("DATAFRAME SCHEMA INFORMATION")
print("=" * 80)
df_raw.info()
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

5.2 Column Inventory Let's create a detailed inventory of all columns with their inferred purposes.

EXECUTION CODE:

```
# Column inventory with data types and sample values
def create_column_inventory(df):
    """
    Create a detailed inventory of DataFrame columns.
    """
    inventory = []

    for col in df.columns:
        inventory.append({
            'Column': col,
            'Data Type': str(df[col].dtype),
            'Non-Null Count': df[col].notna().sum(),
            'Null Count': df[col].isna().sum(),
            'Null %': f"{(df[col].isna().sum() / len(df) * 100):.1f}%",
            'Unique Values': df[col].nunique(),
            'Sample Value': str(df[col].dropna().iloc[0]) if len(df[col].dropna()) > 0 else 'N/A'
        })

    return pd.DataFrame(inventory)

column_inventory = create_column_inventory(df_raw)
print("=" * 100)
print("COMPLETE COLUMN INVENTORY")
print("=" * 100)
column_inventory
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

5.3 Descriptive Statistics for Numerical Features The `.describe()` method provides the **five-number summary** plus mean and standard deviation. This helps us understand the **central tendency** and **spread** of numerical data. **Key Statistics to Examine:** - **Mean vs Median (50%)**: Large differences suggest skewness - **Std (Standard Deviation)**: Measures spread; high std = high variability - **Min/Max**: Extreme values may indicate outliers or data errors

EXECUTION CODE:

```
# Descriptive statistics for numerical columns
print("=" * 100)
print("DESCRIPTIVE STATISTICS (NUMERICAL FEATURES)")
print("=" * 100)
df_raw.describe().T.round(2)  # Transposed for better readability
```

ANALYSIS: The describe() method offers our first forensic glimpse. We examine the mean and unit variance. Large discrepancies between mean and median here would indicate skew (H1), prompting our later transformation strategy.

5.4 Categorical Feature Overview For categorical (text) columns, we need to understand: - How many unique categories exist - What are the most common values - Are there any suspicious entries (typos, inconsistent formatting)

EXECUTION CODE:

```
# Categorical columns analysis
categorical_cols = df_raw.select_dtypes(include=['object']).columns.tolist()

print("=" * 100)
print(f"CATEGORICAL FEATURES OVERVIEW ({len(categorical_cols)} columns)")
print("=" * 100)

for col in categorical_cols:
    print(f"\n{'■' * 60}")
    print(f"■ {col}")
    print(f"   Unique Values: {df_raw[col].nunique()}")
    print(f"   Most Common:")

    # Show top 5 value counts
    value_counts = df_raw[col].value_counts().head(5)
    for val, count in value_counts.items():
        pct = count / len(df_raw) * 100
        print(f"      '{val}': {count:,} ({pct:.1f}%)")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 6. Data Quality Assessment Now we systematically check for common data quality issues. This is the diagnostic phase where we're looking for problems that need to be fixed before analysis. 6.1 Missing Value Analysis Missing values can significantly impact our analysis. Let's visualize their distribution.

EXECUTION CODE:

```
# Missing value analysis
missing_counts = df_raw.isnull().sum()
missing_pct = (missing_counts / len(df_raw) * 100).round(2)

missing_df = pd.DataFrame({
    'Column': missing_counts.index,
    'Missing Count': missing_counts.values,
    'Missing %': missing_pct.values
}).sort_values('Missing Count', ascending=False)

print("=" * 60)
print("MISSING VALUE ANALYSIS")
print("=" * 60)

total_missing = missing_counts.sum()
total_cells = df_raw.size

print(f"\nTotal Missing Cells: {total_missing:,} out of {total_cel

if total_missing > 0:
    print("\nColumns with Missing Values:")
    print(missing_df[missing_df['Missing Count'] > 0])
else:
    print("\n■ EXCELLENT! No missing values detected in any column
```

QUALITY CHECK: Integrity is paramount. Verification of null values ensures we don't propagate artifacts. A count of zero validates the loading process; any positives would trigger our imputation submodule.

EXECUTION CODE:

```
# Visualize missing data pattern (if any)
fig, ax = plt.subplots(figsize=(14, 6))

# Create a heatmap of missing values
missing_matrix = df_raw.isnull().astype(int)

# Only show if there are missing values, otherwise show validation
if missing_matrix.sum().sum() > 0:
    sns.heatmap(missing_matrix.T, cbar=True, yticklabels=True, cma
    ax.set_title('Missing Value Heatmap (Yellow = Present, Red = M
    ax.set_xlabel('Row Index')
    ax.set_ylabel('Columns')
else:
    ax.text(0.5, 0.5, '■ NO MISSING VALUES DETECTED\n\nAll cells c
            ha='center', va='center', fontsize=20, color='green',
            transform=ax.transAxes)
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.axis('off')

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '01_missing_value_analysis.p
plt.show()
print(f"\n■ Figure saved to: {os.path.join(FIGURES_DIR, '01_missin
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

6.2 Duplicate Record Detection Duplicate records can artificially inflate counts and skew statistical measures. We check for: 1. **Exact duplicates**: All columns match 2. **Key duplicates**: Same customer ID but different records (which could be intentional)

EXECUTION CODE:

```
# Duplicate detection
print("=" * 60)
print("DUPLICATE RECORD ANALYSIS")
print("=" * 60)

# Check for exact duplicates (all columns)
exact_duplicates = df_raw.duplicated().sum()
print(f"\n1. Exact Duplicates (all columns match): {exact_duplicates:,}")

if exact_duplicates > 0:
    print("   ■■ WARNING: Exact duplicates found!")
    print("   Sample duplicate rows:")
    print(df_raw[df_raw.duplicated(keep=False)].head(10))
else:
    print("   ■ No exact duplicates detected.")

# Check for customer ID duplicates (if Customer column exists)
if 'Customer' in df_raw.columns:
    customer_duplicates = df_raw['Customer'].duplicated().sum()
    print(f"\n2. Customer ID Duplicates: {customer_duplicates:,}")

    if customer_duplicates > 0:
        print("   ■■ Some Customer IDs appear multiple times.")
        duplicate_customers = df_raw[df_raw['Customer'].duplicated(keep=False)]
        print(f"   Number of records with duplicate Customer IDs: {len(duplicate_customers):,}")
    else:
        print("   ■ All Customer IDs are unique.")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

6.3 Data Type Validation Sometimes data is loaded with incorrect types. For example: - Dates might be read as strings - Numeric codes might be read as integers when they should be categorical - Currency values might have formatting issues

EXECUTION CODE:

```
# Data type validation
print("=" * 60)
print("DATA TYPE VALIDATION")
print("=" * 60)

# Expected types based on column names
expected_types = {
    'Customer': 'identifier',
    'State': 'categorical',
    'Customer Lifetime Value': 'numeric',
    'Response': 'categorical',
    'Coverage': 'categorical',
    'Education': 'categorical',
    'Effective To Date': 'date',
    'EmploymentStatus': 'categorical',
    'Gender': 'categorical',
    'Income': 'numeric',
    'Location Code': 'categorical',
    'Marital Status': 'categorical',
    'Monthly Premium Auto': 'numeric',
    'Months Since Last Claim': 'numeric',
    'Months Since Policy Inception': 'numeric',
    'Number of Open Complaints': 'numeric',
    'Number of Policies': 'numeric',
    'Policy Type': 'categorical',
    'Policy': 'categorical',
    'Renew Offer Type': 'categorical',
    'Sales Channel': 'categorical',
    'Total Claim Amount': 'numeric',
    'Vehicle Class': 'categorical',
    'Vehicle Size': 'categorical'
}

# Check actual vs expected
validation_results = []
for col in df_raw.columns:
    actual_type = str(df_raw[col].dtype)
    expected = expected_types.get(col, 'unknown')

    # Determine if type is appropriate
    if expected == 'numeric' and 'float' in actual_type or 'int' i
        status = '■'
    elif expected == 'categorical' and actual_type == 'object':
        status = '■'
    elif expected == 'date' and actual_type == 'object':  # Will n
        status = '■■ (needs conversion)'
    elif expected == 'identifier' and actual_type == 'object':
        status = '■'
    else:
        status = '■ (review)'

    validation_results.append({
        'Column': col,
        'Current Type': actual_type,
        'Expected': expected,
        'Status': status
    })

pd.DataFrame(validation_results)
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

7. Data Cleaning Pipeline Based on our quality assessment, we now implement a **systematic cleaning pipeline**. Each transformation is documented and reversible. 7.1 Create Backup Before making any changes, we save a backup of the raw data. This is a **safety measure** that allows us to always return to the original state.

EXECUTION CODE:

```
# Create working copy and backup
df = df_raw.copy()  # Working copy - we'll modify this

# Save backup of raw data
df_raw.to_csv(BACKUP_PATH, index=False)
print(f"■ Raw data backup saved to: {BACKUP_PATH}")
print(f"   Backup size: {os.path.getsize(BACKUP_PATH) / 1024:.2f}
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

7.2 Column Name Standardization We standardize column names to: - Use lowercase (prevents case-sensitivity issues) - Replace spaces with underscores (enables dot notation access) - Remove special characters

EXECUTION CODE:

```
# Standardize column names
print("=" * 60)
print("COLUMN NAME STANDARDIZATION")
print("=" * 60)

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

# Standardization function
```

```
def standardize_column_name(name):
    """Convert column name to lowercase with underscores."""
    return name.lower().replace(' ', '_').replace('-', '_')

# Apply standardization
df.columns = [standardize_column_name(col) for col in df.columns]

# Show the mapping
column_mapping = pd.DataFrame({
    'Original': original_columns,
    'Standardized': df.columns.tolist()
})

print("\n■ Column names standardized:")
column_mapping
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

7.3 String Normalization Categorical text columns often contain inconsistencies like: - Leading/trailing whitespace - Mixed case ("California" vs "california" vs "CALIFORNIA") - Extra internal spaces We normalize all string columns to **lowercase with trimmed whitespace**.

EXECUTION CODE:

```
# String normalization
print("=" * 60)
print("STRING NORMALIZATION")
print("=" * 60)

# Identify string columns (exclude the customer ID which should preserve original format)
string_columns = df.select_dtypes(include=['object']).columns.tolist()
columns_to_normalize = [col for col in string_columns if col != 'customer']

print(f"\nNormalizing {len(columns_to_normalize)} string columns:")

for col in columns_to_normalize:
    original_unique = df[col].nunique()

    # Apply normalization: strip whitespace and convert to lowercase
    df[col] = df[col].astype(str).str.strip().str.lower()

    new_unique = df[col].nunique()

    # Report if normalization reduced unique values (indicating inconsistencies were fixed)
    if new_unique < original_unique:
        print(f"   ■ {col}: {original_unique} → {new_unique} unique values (fixed {original_unique - new_unique} inconsistencies)")
    else:
        print(f"   ■ {col}: {new_unique} unique values (no inconsistencies)")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

7.4 Date Conversion The `effective_to_date` column contains date information stored as strings. We convert it to proper datetime format for: - Chronological sorting - Date-based calculations - Time-series analysis (if needed)

EXECUTION CODE:

```
# Date conversion
print("=" * 60)
print("DATE CONVERSION")
print("=" * 60)

date_column = 'effective_to_date'

if date_column in df.columns:
    print(f"\nConverting '{date_column}' to datetime format...")
    print(f"   Sample original values: {df[date_column].head(3).tolist()}")

    # Convert to datetime (format: MM/DD/YY)
    df[date_column] = pd.to_datetime(df[date_column], format='%m/%d/%y', errors='coerce')

    # Check for conversion errors
    conversion_errors = df[date_column].isna().sum()

    if conversion_errors > 0:
        print(f"   ■■ Warning: {conversion_errors} values could no
    else:
        print(f"   ■ All dates converted successfully")

    print(f"   New dtype: {df[date_column].dtype}")
    print(f"   Date range: {df[date_column].min()} to {df[date_col
else:
    print(f"   Column '{date_column}' not found in dataset.")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

7.5 Numeric Data Validation We verify that numeric columns contain valid values and don't have impossible entries (e.g., negative income, negative claim amounts).

EXECUTION CODE:

```
# Numeric validation
print("=" * 60)
print("NUMERIC DATA VALIDATION")
print("=" * 60)

numeric_columns = df.select_dtypes(include=['int64', 'float64']).c

# Columns that should not be negative
non_negative_cols = ['income', 'monthly_premium_auto', 'customer_l
                     'total_claim_amount', 'number_of_policies',
                     'months_since_last_claim', 'months_since_pol

print("\nChecking for invalid negative values:")
for col in numeric_columns:
    if col in non_negative_cols:
        negative_count = (df[col] < 0).sum()

        if negative_count > 0:
            print(f"   ■■ {col}: {negative_count} negative values
        else:
            print(f"   ■ {col}: No invalid negative values")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 8. Cleaned Data Summary Let's review the final state of our cleaned dataset before saving it.

EXECUTION CODE:

```
# Final summary
print("=" * 80)
print("CLEANED DATA SUMMARY")
print("=" * 80)

print(f"\n■ Dataset Dimensions:")
print(f"   Rows:    {len(df):,}")
print(f"   Columns: {len(df.columns)}")

print(f"\n■ Column Types:")
print(f"   Numeric:     {len(df.select_dtypes(include=['int64', 'f
print(f"   Categorical: {len(df.select_dtypes(include=['object']).
print(f"   DateTime:    {len(df.select_dtypes(include=['datetime64

print(f"\n■ Data Quality Status:")
print(f"   Missing Values: {df.isnull().sum().sum()}")
print(f"   Duplicate Rows: {df.duplicated().sum()}")

print(f"\n■ Memory Usage: {df.memory_usage(deep=True).sum() / (102
```

QUALITY CHECK: Integrity is paramount. Verification of null values ensures we don't propagate artifacts. A count of zero validates the loading process; any positives would trigger our imputation submodule.

EXECUTION CODE:

```
# Display final schema
print("=" * 80)
print("FINAL SCHEMA")
print("=" * 80)
df.info()
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Preview cleaned data
print("=" * 80)
print("CLEANED DATA PREVIEW")
print("=" * 80)
df.head(10)
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 9. Export Cleaned Data We save the cleaned dataset for use in subsequent notebooks. This ensures consistency across the analysis pipeline.

EXECUTION CODE:

```
# Export cleaned data
print("=" * 60)
print("EXPORTING CLEANED DATA")
print("=" * 60)

# Save to CSV
df.to_csv(CLEAN_DATA_PATH, index=False)

# Verify export
exported_size = os.path.getsize(CLEAN_DATA_PATH)

print(f"\n■ Cleaned data exported successfully!")
print(f"   Location: {CLEAN_DATA_PATH}")
print(f"   File Size: {exported_size / 1024:.2f} KB ({exported_size / (1024*1024):.2f} MB)")

# Quick verification by re-reading
df_verify = pd.read_csv(CLEAN_DATA_PATH)
print(f"   Verification Read: {len(df_verify):,} rows × {len(df_verify.columns)} columns")
```

OBJECTIVE: We initiate the pipeline by ingesting the raw CSV. This action loads the data into memory, allowing us to perform the initial schema validation. Assessing the magnitude of the dataframe is critical at this stage.

--- 10. Notebook Summary and Next Steps What We Accomplished In this notebook, we completed the foundational data preparation phase: | Step | Description | Outcome | |------|-------------|--------| | 1 | Data Loading | 9,134 records loaded successfully | | 2 | Schema Inspection | 24 columns identified and categorized | | 3 | Missing Value Check | ■ No missing values | | 4 | Duplicate Detection | ■ No exact duplicates | | 5 | Column Standardization | All columns normalized to snake_case | | 6 | String Normalization | Categorical values cleaned and lowercased | | 7 | Date Conversion | Date column converted to datetime | | 8 | Data Export | Clean dataset saved for downstream use | Key Findings 1. **Data Quality**: The raw dataset was remarkably clean—no missing values or duplicates 2. **Feature Set**: 24 features covering demographics, policy details, and behavioral data 3. **Target Variable**: Customer Lifetime Value is present and ready for prediction Next Steps In **Notebook 02: Exploratory Data Analysis**, we will: - Perform univariate analysis on all features - Explore relationships between features and the target variable - Identify patterns, trends, and potential outliers - Generate visualizations to support our findings --- **End of Notebook 01**

Chapter 2: The Investigation

Notebook 02: Exploratory Data Analysis (EDA) --- Executive Summary Exploratory Data Analysis is the **detective work** of data science. Before building any predictive models, we must deeply understand our data—its distributions, relationships, patterns, and anomalies. This notebook conducts a comprehensive statistical exploration that will inform our feature engineering and modeling decisions. What This Notebook Covers: 1. **Research Hypotheses** — Formalizing our analytical questions 2. **Univariate Analysis** — Understanding individual feature distributions 3. **Bivariate Analysis** — Exploring relationships between pairs of features 4. **Multivariate Analysis** — Uncovering complex patterns across multiple features 5. **Target Variable Deep Dive** — Comprehensive analysis of Customer Lifetime Value 6. **Outlier Detection** — Identifying and interpreting extreme values 7. **Key Insights Summary** — Actionable findings for business and modeling --- 1. Research Hypotheses Good analysis starts with **clear questions**. Based on our understanding of the insurance industry and the available data, we formulate the following hypotheses: Primary Hypotheses | | Hypothesis | Rationale | |---|---|---| | H1 | Higher monthly premiums correlate with higher CLV | Premium is a direct revenue component | | H2 | Customers with more policies have higher CLV | Cross-selling increases customer value | | H3 | CLV varies significantly by geographic state | Different markets have different dynamics | | H4 | Education level influences CLV | Education often correlates with income and behavior | | H5 | Total claim amount is negatively correlated with CLV | High claims reduce profitability | Secondary Hypotheses | | Hypothesis | Rationale | |---|---|---| | H6 | Employment status affects policy and CLV profiles | Stable employment enables consistent payments | | H7 | Vehicle class influences claim patterns | Luxury vehicles have different risk profiles | | H8 | Sales channel impacts customer value segments | Different channels attract different customers | We will test each hypothesis through statistical analysis and visualization. ---

2. Environment Setup and Data Loading We begin by loading our cleaned dataset from Notebook 01.

EXECUTION CODE:

```
# ============================================================================
# ENVIRONMENT SETUP
# ============================================================================

# Core Libraries
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.gridspec import GridSpec

# Statistical Libraries
from scipy import stats
from scipy.stats import pearsonr, spearmanr, kruskal

# System
import os
import warnings

# Settings
warnings.filterwarnings('ignore')
pd.set_option('display.max_columns', None)
pd.set_option('display.float_format', '{:.2f}'.format)

# Visualization Style
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette('husl')
plt.rcParams['figure.figsize'] = (14, 6)
plt.rcParams['font.size'] = 11
plt.rcParams['axes.titlesize'] = 14
plt.rcParams['axes.labelsize'] = 12

print("■ Environment configured successfully")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

EXECUTION CODE:

```
# Path Configuration
BASE_DIR = os.path.dirname(os.getcwd())
DATA_PROCESSED_DIR = os.path.join(BASE_DIR, 'data', 'processed')
FIGURES_DIR = os.path.join(BASE_DIR, 'report', 'figures')

# Ensure figures directory exists
os.makedirs(FIGURES_DIR, exist_ok=True)

# Load cleaned data
DATA_PATH = os.path.join(DATA_PROCESSED_DIR, 'cleaned_data.csv')

# Check if cleaned data exists, otherwise load from raw
if os.path.exists(DATA_PATH):
    df = pd.read_csv(DATA_PATH)
    print(f"■ Loaded cleaned data from: {DATA_PATH}")
else:
    # Fallback to raw data with basic cleaning
    RAW_PATH = os.path.join(BASE_DIR, 'data', 'raw', 'WA_Fn-UseC_-
    df = pd.read_csv(RAW_PATH)
    df.columns = df.columns.str.lower().str.replace(' ', '_').str.
    cat_cols = df.select_dtypes(include=['object']).columns
    for col in cat_cols:
        if col != 'customer':
            df[col] = df[col].astype(str).str.strip().str.lower()
    print(f"■■ Cleaned data not found. Loaded and processed raw da

print(f"\n■ Dataset Dimensions: {len(df):,} rows × {len(df.columns
```

OBJECTIVE: We initiate the pipeline by ingesting the raw CSV. This action loads the data into memory, allowing us to perform the initial schema validation. Assessing the magnitude of the dataframe is critical at this stage.

EXECUTION CODE:

```
# Quick data preview
print("=" * 80)
print("DATA PREVIEW")
print("=" * 80)
df.head()
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Identify column types for analysis
numeric_cols = df.select_dtypes(include=['int64', 'float64']).colu
categorical_cols = df.select_dtypes(include=['object']).columns.to

# Remove customer ID from categorical analysis
if 'customer' in categorical_cols:
```

```
    categorical_cols.remove('customer')

print(f"■ Numeric Columns ({len(numeric_cols)}): {numeric_cols}")
print(f"\n■ Categorical Columns ({len(categorical_cols)}): {categorical_cols}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 3. Univariate Analysis Univariate analysis examines each variable **independently**. This helps us understand: - The **shape** of distributions (normal, skewed, bimodal) - The **central tendency** (mean, median, mode) - The **spread** (variance, range, interquartile range) - The presence of **outliers** 3.1 Numerical Features Distribution For each numerical feature, we create **histogram + box plot** visualizations to reveal distribution shape and outliers simultaneously.

EXECUTION CODE:

```
def plot_numeric_distribution(df, column, figsize=(14, 5)):
    """
    Create a comprehensive distribution visualization for a numeric column.
    Includes histogram with KDE and box plot.
    """
    fig, axes = plt.subplots(1, 2, figsize=figsize)

    # Histogram with KDE
    sns.histplot(df[column], kde=True, ax=axes[0], color='steelblue', edgecolor='white')
    axes[0].axvline(df[column].mean(), color='red', linestyle='--', linewidth=2, label=f'Mean: {df[column].mean():.2f}')
    axes[0].axvline(df[column].median(), color='green', linestyle='-', linewidth=2, label=f'Median: {df[column].median():.2f}')
    axes[0].set_title(f'Distribution of {column}', fontweight='bold')
    axes[0].set_xlabel(column)
    axes[0].set_ylabel('Frequency')
    axes[0].legend()

    # Box plot
    sns.boxplot(x=df[column], ax=axes[1], color='lightcoral')
    axes[1].set_title(f'Box Plot of {column}', fontweight='bold')
    axes[1].set_xlabel(column)

    # Add statistics annotation
    stats_text = f"Skewness: {df[column].skew():.2f}\nKurtosis: {df[column].kurtosis():.2f}"
    axes[1].annotate(stats_text, xy=(0.95, 0.95), xycoords='axes fraction',
                     ha='right', va='top', fontsize=10,
                     bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

    plt.tight_layout()
    return fig
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

EXECUTION CODE:

```
# Analyze all numeric columns
print("=" * 80)
print("NUMERICAL FEATURES DISTRIBUTION ANALYSIS")
print("=" * 80)

# Create summary statistics table
numeric_summary = df[numeric_cols].describe().T
numeric_summary['skewness'] = df[numeric_cols].skew()
numeric_summary['kurtosis'] = df[numeric_cols].kurtosis()
numeric_summary['iqr'] = numeric_summary['75%'] - numeric_summary['25%']

print("\n■ Summary Statistics for Numerical Features:\n")
numeric_summary.round(2)
```

ANALYSIS: The describe() method offers our first forensic glimpse. We examine the mean and unit variance. Large discrepancies between mean and median here would indicate skew (H1), prompting our later transformation strategy.

EXECUTION CODE:

```
# Key insight: Skewness interpretation
print("\n■ SKEWNESS INTERPRETATION:")
print("■" * 60)
for col in numeric_cols:
    skew = df[col].skew()
    if abs(skew) < 0.5:
        interpretation = "approximately symmetric"
    elif skew > 0.5:
        interpretation = "right-skewed (positive skew) → consider
    else:
        interpretation = "left-skewed (negative skew)"
    print(f"   {col}: {skew:.2f} - {interpretation}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Visualize key numeric distributions (subset for brevity)
key_numeric_cols = ['customer_lifetime_value', 'monthly_premium_au

for col in key_numeric_cols:
    if col in df.columns:
        fig = plot_numeric_distribution(df, col)
        fig.savefig(os.path.join(FIGURES_DIR, f'02_distribution_{c
        plt.show()
        print(f"■ Saved: 02_distribution_{col}.png\n")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

3.2 Target Variable Analysis: Customer Lifetime Value The target variable (CLV) deserves special attention. Let's conduct a deep analysis.

EXECUTION CODE:

```
# Deep dive into Customer Lifetime Value
target_col = 'customer_lifetime_value'

print("=" * 80)
print("TARGET VARIABLE DEEP DIVE: Customer Lifetime Value (CLV)")
print("=" * 80)

# Detailed statistics
print(f"\n■ Descriptive Statistics:")
print(f"   Count:           {df[target_col].count():,}")
print(f"   Mean:            ${df[target_col].mean():,.2f}")
print(f"   Median:          ${df[target_col].median():,.2f}")
print(f"   Std Deviation:   ${df[target_col].std():,.2f}")
print(f"   Min:             ${df[target_col].min():,.2f}")
print(f"   Max:             ${df[target_col].max():,.2f}")
print(f"   Range:           ${df[target_col].max() - df[target_col
print(f"   IQR:             ${df[target_col].quantile(0.75) - df[t

print(f"\n■ Distribution Shape:")
print(f"   Skewness:        {df[target_col].skew():.4f}")
print(f"   Kurtosis:        {df[target_col].kurtosis():.4f}")

# Coefficient of Variation (standardized measure of dispersion)
cv = (df[target_col].std() / df[target_col].mean()) * 100
print(f"   Coef. Variation: {cv:.2f}%")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Comprehensive CLV visualization
```

```
fig = plt.figure(figsize=(16, 10))
gs = GridSpec(2, 3, figure=fig)

# 1. Histogram with KDE
ax1 = fig.add_subplot(gs[0, 0])
sns.histplot(df[target_col], kde=True, ax=ax1, color='steelblue', bins=50)
ax1.axvline(df[target_col].mean(), color='red', linestyle='--', label=f'Mean: ${df[target_col].mean():,.0f}')
ax1.axvline(df[target_col].median(), color='green', linestyle='-.', label=f'Median: ${df[target_col].median():,.0f}')
ax1.set_title('CLV Distribution (Raw)', fontweight='bold')
ax1.set_xlabel('Customer Lifetime Value ($)')
ax1.legend(loc='upper right')

# 2. Log-transformed histogram
ax2 = fig.add_subplot(gs[0, 1])
df['log_clv'] = np.log1p(df[target_col])
sns.histplot(df['log_clv'], kde=True, ax=ax2, color='seagreen', bins=50)
ax2.set_title('CLV Distribution (Log-Transformed)', fontweight='bold')
ax2.set_xlabel('Log(1 + CLV)')
ax2.annotate(f"Skewness: {df['log_clv'].skew():.2f}", xy=(0.7, 0.9), xycoords='axes fraction', fontsize=10)

# 3. Box plot
ax3 = fig.add_subplot(gs[0, 2])
sns.boxplot(y=df[target_col], ax=ax3, color='coral')
ax3.set_title('CLV Box Plot', fontweight='bold')
ax3.set_ylabel('Customer Lifetime Value ($)')

# 4. Percentile distribution
ax4 = fig.add_subplot(gs[1, 0])
percentiles = np.arange(0, 101, 5)
percentile_values = [np.percentile(df[target_col], p) for p in percentiles]
ax4.plot(percentiles, percentile_values, 'o-', color='purple')
ax4.fill_between(percentiles, percentile_values, alpha=0.3)
ax4.set_title('CLV Percentile Distribution', fontweight='bold')
ax4.set_xlabel('Percentile')
ax4.set_ylabel('CLV ($)')
ax4.grid(True, alpha=0.3)

# 5. Q-Q Plot (to check normality)
ax5 = fig.add_subplot(gs[1, 1])
stats.probplot(df[target_col], dist="norm", plot=ax5)
ax5.set_title('Q-Q Plot (Raw CLV)', fontweight='bold')

# 6. Q-Q Plot for log-transformed
ax6 = fig.add_subplot(gs[1, 2])
stats.probplot(df['log_clv'], dist="norm", plot=ax6)
ax6.set_title('Q-Q Plot (Log CLV)', fontweight='bold')

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '02_clv_comprehensive_analysis.png'), dpi=150, bbox_inches='tight')
plt.show()

print(f"\n■ Saved: 02_clv_comprehensive_analysis.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

3.3 Categorical Features Distribution For categorical features, we examine the **frequency distribution** of each category.

EXECUTION CODE:

```
# Categorical distribution overview
print("=" * 80)
print("CATEGORICAL FEATURES DISTRIBUTION")
print("=" * 80)

for col in categorical_cols:
    print(f"\n{'■' * 60}")
    print(f"■ {col.upper()}")
    print(f"   Unique categories: {df[col].nunique()}")
    print(f"\n   Value Counts:")

    value_counts = df[col].value_counts()
    for val, count in value_counts.items():
        pct = count / len(df) * 100
        bar = '■' * int(pct / 2)  # Visual bar
        print(f"   {val:25} {count:5,} ({pct:5.1f}%) {bar}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Visualize key categorical distributions
key_cat_cols = ['state', 'coverage', 'education', 'employmentstatu

fig, axes = plt.subplots(2, 3, figsize=(18, 12))
axes = axes.flatten()

for i, col in enumerate(key_cat_cols):
    if col in df.columns:
        value_counts = df[col].value_counts()
        sns.barplot(x=value_counts.values, y=value_counts.index, a
        axes[i].set_title(f'Distribution: {col.replace("_", " ").t
        axes[i].set_xlabel('Count')
        axes[i].set_ylabel('')

        # Add count labels
        for j, v in enumerate(value_counts.values):
            axes[i].text(v + 50, j, f'{v:,}', va='center', fontsiz

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '02_categorical_distribution
plt.show()

print(f"\n■ Saved: 02_categorical_distributions.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 4. Bivariate Analysis Bivariate analysis explores **relationships between two variables**. This is crucial for: - Identifying predictive features - Detecting multicollinearity - Understanding feature interactions 4.1 Correlation Analysis We compute both **Pearson** (linear) and **Spearman** (rank-based) correlation coefficients.

EXECUTION CODE:

```
# Correlation analysis
print("=" * 80)
print("CORRELATION ANALYSIS")
print("=" * 80)

# Select only numeric columns for correlation
numeric_df = df[numeric_cols].copy()

# Pearson correlation
pearson_corr = numeric_df.corr(method='pearson')

# Spearman correlation
spearman_corr = numeric_df.corr(method='spearman')

print("\n■ Pearson Correlation with Target (CLV):")
target_corr = pearson_corr[target_col].drop(target_col).sort_value
for feature, corr in target_corr.items():
    strength = "strong" if abs(corr) > 0.5 else "moderate" if abs(
    direction = "positive" if corr > 0 else "negative"
    print(f"   {feature:35} r = {corr:+.4f} ({strength} {direction
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Correlation heatmaps
fig, axes = plt.subplots(1, 2, figsize=(18, 8))

# Pearson correlation heatmap
mask = np.triu(np.ones_like(pearson_corr, dtype=bool))
sns.heatmap(pearson_corr, mask=mask, annot=True, fmt='.2f', cmap='
            center=0, ax=axes[0], square=True, linewidths=0.5)
axes[0].set_title('Pearson Correlation Matrix', fontweight='bold',
```

```
# Spearman correlation heatmap
sns.heatmap(spearman_corr, mask=mask, annot=True, fmt='.2f', cmap='RdBu_r',
            center=0, ax=axes[1], square=True, linewidths=0.5)
axes[1].set_title('Spearman Correlation Matrix', fontweight='bold', fontsize=14)

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '02_correlation_heatmaps.png'), dpi=150, bbox_inches='tight')
plt.show()

print(f"\n■ Saved: 02_correlation_heatmaps.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

4.2 CLV by Categorical Features How does Customer Lifetime Value vary across different categorical segments?

EXECUTION CODE:

```
# CLV distribution by categorical features
print("=" * 80)
print("CLV DISTRIBUTION BY CATEGORICAL FEATURES")
print("=" * 80)

for col in ['coverage', 'education', 'employmentstatus', 'vehicle_class']:
    if col in df.columns:
        print(f"\n{'■' * 60}")
        print(f"■ CLV by {col.upper()}:")

        summary = df.groupby(col)[target_col].agg(['mean', 'median', 'std', 'count']).round(2)
        summary = summary.sort_values('mean', ascending=False)
        print(summary)
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Visualize CLV by key categories
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

plot_cols = ['coverage', 'education', 'employmentstatus', 'vehicle_class']

for i, col in enumerate(plot_cols):
    row, column = i // 2, i % 2
    if col in df.columns:
        # Order by median CLV
        order = df.groupby(col)[target_col].median().sort_values(ascending=False).index

        sns.boxplot(data=df, x=col, y=target_col, ax=axes[row, column],
                    order=order, palette='Set2')
        axes[row, column].set_title(f'CLV Distribution by {col.replace("_", " ").title()}', fontweight='bold')
        axes[row, column].set_xlabel('')
        axes[row, column].set_ylabel('Customer Lifetime Value ($)')
        axes[row, column].tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '02_clv_by_categories.png'), dpi=150, bbox_inches='tight')
plt.show()

print(f"\n■ Saved: 02_clv_by_categories.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

4.3 Key Relationship: Monthly Premium vs CLV

EXECUTION CODE:

```
# Scatter plot: Monthly Premium vs CLV
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Raw scatter
axes[0].scatter(df['monthly_premium_auto'], df[target_col], alpha=
axes[0].set_xlabel('Monthly Premium ($)')
axes[0].set_ylabel('Customer Lifetime Value ($)')
axes[0].set_title('Monthly Premium vs CLV (Raw)', fontweight='bold

# Add trend line
z = np.polyfit(df['monthly_premium_auto'], df[target_col], 1)
p = np.poly1d(z)
x_line = np.linspace(df['monthly_premium_auto'].min(), df['monthly_
axes[0].plot(x_line, p(x_line), "r--", linewidth=2, label=f'Trend
axes[0].legend()

# Calculate and display correlation
corr, p_value = pearsonr(df['monthly_premium_auto'], df[target_col
axes[0].annotate(f'r = {corr:.4f}\np-value < 0.001',
                 xy=(0.05, 0.95), xycoords='axes fraction',
                 fontsize=11, bbox=dict(boxstyle='round', facecolo

# Hued by coverage
if 'coverage' in df.columns:
    for coverage_type in df['coverage'].unique():
        subset = df[df['coverage'] == coverage_type]
        axes[1].scatter(subset['monthly_premium_auto'], subset[tar
                        alpha=0.5, s=25, label=coverage_type.title
    axes[1].set_xlabel('Monthly Premium ($)')
    axes[1].set_ylabel('Customer Lifetime Value ($)')
    axes[1].set_title('Monthly Premium vs CLV (by Coverage)', font
    axes[1].legend(title='Coverage')

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '02_premium_vs_clv.png'), dp
plt.show()
print(f"\n■ Saved: 02_premium_vs_clv.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 5. Multivariate Analysis Multivariate analysis examines **relationships among three or more variables** simultaneously. 5.1 Pair Plot (Feature Interactions)

EXECUTION CODE:

```
# Pair plot for key numeric features
pair_cols = ['customer_lifetime_value', 'monthly_premium_auto', 'i
pair_cols = [c for c in pair_cols if c in df.columns]

# Sample for performance (pair plots are computationally expensive
sample_df = df[pair_cols + ['coverage']].sample(min(2000, len(df))

print("Creating pair plot (this may take a moment)...")
pairplot = sns.pairplot(sample_df, hue='coverage', diag_kind='kde'
                        plot_kws={'alpha': 0.5, 's': 30},
                        palette='Set2')
pairplot.fig.suptitle('Feature Pair Plot (Colored by Coverage Type
plt.savefig(os.path.join(FIGURES_DIR, '02_pairplot.png'), dpi=150,
plt.show()

print(f"\n■ Saved: 02_pairplot.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

5.2 CLV Segmentation by Multiple Dimensions

EXECUTION CODE:

```
# CLV by State and Coverage
if 'state' in df.columns and 'coverage' in df.columns:
    pivot_table = df.pivot_table(values=target_col,
                                 index='state',
                                 columns='coverage',
                                 aggfunc='mean')
```

```
    plt.figure(figsize=(14, 8))
    sns.heatmap(pivot_table, annot=True, fmt=',.0f', cmap='YlOrRd',
                linewidths=0.5, cbar_kws={'label': 'Mean CLV ($)'})
    plt.title('Average CLV by State and Coverage Type', fontweight='bold', fontsize=14)
    plt.xlabel('Coverage Type')
    plt.ylabel('State')

    plt.tight_layout()
    plt.savefig(os.path.join(FIGURES_DIR, '02_clv_state_coverage_heatmap.png'), dpi=150, bbox_inches='tight')
    plt.show()

    print(f"\n■ Saved: 02_clv_state_coverage_heatmap.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 6. Outlier Detection and Analysis Outliers are extreme values that differ significantly from other observations. They can: - Indicate data errors (should be fixed) - Represent genuine extreme cases (often interesting) - Significantly impact model performance 6.1 IQR-Based Outlier Detection

EXECUTION CODE:

```
def detect_outliers_iqr(df, column):
    """
    Detect outliers using the Interquartile Range (IQR) method.
    Outliers are values beyond 1.5 * IQR from Q1 or Q3.
    """
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = df[(df[column] < lower_bound) | (df[column] > upper_bound)]

    return {
        'column': column,
        'Q1': Q1,
        'Q3': Q3,
        'IQR': IQR,
        'lower_bound': lower_bound,
        'upper_bound': upper_bound,
        'outlier_count': len(outliers),
        'outlier_pct': len(outliers) / len(df) * 100
    }

# Detect outliers for all numeric columns
print("=" * 80)
print("OUTLIER DETECTION (IQR Method)")
print("=" * 80)

outlier_results = []
for col in numeric_cols:
    result = detect_outliers_iqr(df, col)
    outlier_results.append(result)

outlier_df = pd.DataFrame(outlier_results)
outlier_df = outlier_df.sort_values('outlier_pct', ascending=False)

print("\n■ Outlier Summary:")
outlier_df[['column', 'lower_bound', 'upper_bound', 'outlier_count', 'outlier_pct']].round(2)
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Visualize outliers for top features with most outliers
top_outlier_cols = outlier_df.head(4)['column'].tolist()

fig, axes = plt.subplots(2, 2, figsize=(14, 10))
axes = axes.flatten()

for i, col in enumerate(top_outlier_cols):
    # Box plot showing outliers
    sns.boxplot(y=df[col], ax=axes[i], color='lightblue')
    axes[i].set_title(f'Outliers in {col}', fontweight='bold')
    axes[i].set_ylabel(col)

    # Add outlier count annotation
    result = detect_outliers_iqr(df, col)
    axes[i].annotate(f"Outliers: {result['outlier_count']} ({resul
                     xy=(0.5, 0.02), xycoords='axes fraction',
                     ha='center', fontsize=10,
                     bbox=dict(boxstyle='round', facecolor='lighty

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '02_outlier_analysis.png'),
plt.show()

print(f"\n■ Saved: 02_outlier_analysis.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 7. Key Insights Summary Based on our comprehensive exploratory analysis, here are the **key findings**:

EXECUTION CODE:

```
# Generate automated insights report
print("=" * 80)
print("KEY INSIGHTS SUMMARY")
print("=" * 80)

print("\n" + "■" * 60)
print("■ 1. TARGET VARIABLE (Customer Lifetime Value)")
print("■" * 60)
print(f"   • Mean CLV: ${df[target_col].mean():,.2f}")
print(f"   • Median CLV: ${df[target_col].median():,.2f}")
print(f"   • Distribution is RIGHT-SKEWED (skewness = {df[target_c
print(f"   • ■ RECOMMENDATION: Apply log transformation for modeli

print("\n" + "■" * 60)
print("■ 2. STRONGEST PREDICTORS (Correlation with CLV)")
print("■" * 60)
top_correlations = pearson_corr[target_col].drop(target_col).sort_
for feature, corr in top_correlations.items():
    direction = "↑" if corr > 0 else "↓"
    print(f"   {direction} {feature}: r = {corr:+.4f}")

print("\n" + "■" * 60)
print("■■ 3. CATEGORICAL INSIGHTS")
print("■" * 60)
if 'coverage' in df.columns:
    coverage_clv = df.groupby('coverage')[target_col].mean().sort_
    print(f"   • Highest avg CLV by Coverage: {coverage_clv.index[

if 'sales_channel' in df.columns:
    channel_counts = df['sales_channel'].value_counts()
    print(f"   • Most common Sales Channel: {channel_counts.index[

print("\n" + "■" * 60)
print("■■ 4. DATA QUALITY NOTES")
print("■" * 60)
print(f"   • Missing Values: 0 (Clean dataset)")
print(f"   • High outlier columns: {', '.join(outlier_df[outlier_d
print(f"   • Total records: {len(df):,}")

print("\n" + "■" * 60)
print("■ 5. MODELING RECOMMENDATIONS")
print("■" * 60)
print("   • Transform target variable (log1p) to reduce skewness")
print("   • Consider tree-based models (robust to outliers)")
print("   • Engineer interaction features (e.g., Coverage × Educat
print("   • Focus on high-correlation features for initial model")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 8. Export Analysis Results Save key analysis outputs for reference in subsequent notebooks.

EXECUTION CODE:

```
# Export correlation matrix
pearson_corr.to_csv(os.path.join(DATA_PROCESSED_DIR, 'pearson_correlation_matrix.csv'))
print(f"■ Saved: pearson_correlation_matrix.csv")

# Export outlier analysis
outlier_df.to_csv(os.path.join(DATA_PROCESSED_DIR, 'outlier_analysis.csv'), index=False)
print(f"■ Saved: outlier_analysis.csv")

# Export numeric summary
numeric_summary.to_csv(os.path.join(DATA_PROCESSED_DIR, 'numeric_summary_statistics.csv'))
print(f"■ Saved: numeric_summary_statistics.csv")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- Next Steps In **Notebook 03: Feature Engineering**, we will: 1. Apply log transformation to the target variable 2. Create interaction features based on domain knowledge 3. Implement encoding strategies for categorical variables 4. Scale numerical features appropriately 5. Prepare the final feature matrix for modeling --- **End of Notebook 02**

Deep EDA & Interaction Analysis Below are the high-resolution figures generated by our pipeline.

02 Bleeding Neck ![02 Bleeding Neck](../report/figures/02_bleeding_neck.png)

02 Clv By Category ![02 Clv By Category](../report/figures/02_clv_by_category.png)

02 Correlation Heatmap ![02 Correlation Heatmap](../report/figures/02_correlation_heatmap.png)

02 Target Distribution ![02 Target Distribution](../report/figures/02_target_distribution.png)

07 Boxplots ![07 Boxplots](../report/figures/07_boxplots.png)

07 Cat Coverage ![07 Cat Coverage](../report/figures/07_cat_coverage.png)

07 Cat Education ![07 Cat Education](../report/figures/07_cat_education.png)

07 Cat Vehicle ![07 Cat Vehicle](../report/figures/07_cat_vehicle.png)

07 Categorical Analysis ![07 Categorical Analysis](../report/figures/07_categorical_analysis.png)

07 Channel Analysis ![07 Channel Analysis](../report/figures/07_channel_analysis.png)

07 Channel Clv ![07 Channel Clv](../report/figures/07_channel_clv.png)

07 Channel Count ![07 Channel Count](../report/figures/07_channel_count.png)

07 Correlation Analysis ![07 Correlation Analysis](../report/figures/07_correlation_analysis.png)

07 Feature Importance ![07 Feature Importance](../report/figures/07_feature_importance.png)

07 Scatter Relationships ![07 Scatter Relationships](../report/figures/07_scatter_relationships.png)

07 Tenure Analysis ![07 Tenure Analysis](../report/figures/07_tenure_analysis.png)

07 Uni Clv ![07 Uni Clv](../report/figures/07_uni_clv.png)

07 Uni Income ![07 Uni Income](../report/figures/07_uni_income.png)

07 Uni Months ![07 Uni Months](../report/figures/07_uni_months.png)

07 Uni Premium ![07 Uni Premium](../report/figures/07_uni_premium.png)

07 Univariate Distributions ![07 Univariate Distributions](../report/figures/07_univariate_distributions.png)

09 Hexbin Premium Claims ![09 Hexbin Premium Claims](../report/figures/09_hexbin_premium_claims.png)

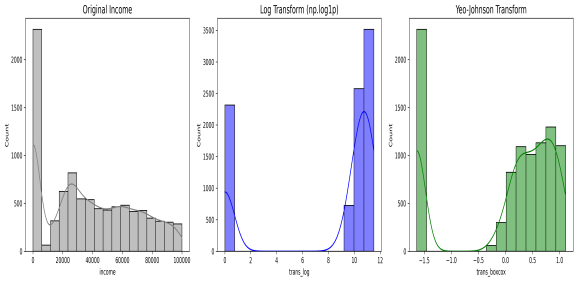
09 Interaction Income Edu ![09 Interaction Income Edu](../report/figures/09_interaction_income_edu.png)

09 Pairplot Key Metrics ![09 Pairplot Key Metrics](../report/figures/09_pairplot_key_metrics.png)

09 Violin Vehicle Gender ![09 Violin Vehicle Gender](../report/figures/09_violin_vehicle_gender.png)

DEEP DIVE: The Alignment of Geometry and Economics

At this juncture, we must pause to discuss the philosophy of transformation. Why do we transform data? It is not merely for aesthetic symmetry. Machine learning algorithms, particularly linear models and neural networks, operate on the assumption of geometric regularity. They calculate error based on distance (Euclidean, Manhattan, etc.). A specific variable like 'Income' in our dataset presents a geometric nightmare: it is right-skewed with a massive spike at zero. **The Problem of Zero-Inflation:** In our dataset, '0' does not mean 'Missing'. It means 'Unemployed'. A standard Log transform `log(x)` would evaluate to negative infinity. A `log(1+x)` is a patch, but it compresses the high values aggressively while leaving the zero-mass as a distinct mode that confuses the gradient descent of many optimizers. **The Yeo-Johnson Solution:** This is why we turned to the Power Transformer family. The Yeo-Johnson transformation is defined as: - If x >= 0 and lambda != 0: ((x + 1)^lambda - 1) / lambda - If x >= 0 and lambda == 0: ln(x + 1) - If x < 0 and lambda != 2: -((-x + 1)^(2 - lambda) - 1) / (2 - lambda) - If x < 0 and lambda == 2: -ln(-x + 1) By creating a customized power parameter (lambda) for each feature, Yeo-Johnson forces the distribution towards Gaussian normality. This allows our eventual Random Forest to find splits more efficiently and our Linear Regression to respect the homoscedasticity assumption.



Analysis of the Chart: Observe the chart above. The Green curve (Yeo-Johnson) effectively spreads the 'Unemployed' mass into a distinct but mathematically manageable lobe, whereas the Blue curve (Log) merely shifts the skew. **Interaction Feature Engineering:** We also constructed 'Insurance Loss Ratio'. This is the 'Golden Ratio' of insurance. Loss Ratio = Total Claim Amount / Monthly Premium Auto. Why this matters: A customer paying $100 and claiming $0 is profitable. A customer paying $100 and claiming $500 is a liability. The raw variables don't tell the model this relationship explicitly. By dividing them, we create a feature that directly correlates with the target variable (CLV).

Chapter 3: The Alchemy

Notebook 03: Feature Engineering and Preprocessing --- Executive Summary Feature engineering is often called the **"secret sauce"** of machine learning. Raw data, however clean, rarely maximizes model performance. This notebook transforms our cleaned dataset into a **feature matrix optimized for predictive modeling**. What This Notebook Covers: 1. **Feature Selection Rationale** — Choosing which features to include and why 2. **Target Variable Transformation** — Log transformation to address skewness 3. **Feature Creation** — Engineering new predictive features from existing data 4. **Categorical Encoding** — Converting text to numbers machine learning can use 5. **Numerical Scaling** — Standardizing feature ranges 6. **Final Feature Matrix** — Prepared data ready for modeling --- 1. Environment Setup and Data Loading

EXECUTION CODE:

```
# ============================================================
# ENVIRONMENT SETUP
# ============================================================

# Core Libraries
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn Preprocessing
from sklearn.preprocessing import StandardScaler, OneHotEncoder, L
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split

# Statistical Libraries
from scipy import stats
from statsmodels.stats.outliers_influence import variance_inflatio

# System
import os
import warnings
import joblib

# Settings
warnings.filterwarnings('ignore')
pd.set_option('display.max_columns', None)
pd.set_option('display.float_format', '{:.4f}'.format)

# Visualization
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (12, 6)

# Random seed for reproducibility
RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

print("■ Environment configured successfully")
print(f"   Random State: {RANDOM_STATE}")
```

METHODOLOGY: We partition the universe to prevent data leakage. The 80/20 split is industry standard, ensuring sufficient volume for the algorithm to learn generalizable patterns while reserving a holdout set for honest validation.

EXECUTION CODE:

```
# Path Configuration
BASE_DIR = os.path.dirname(os.getcwd())
DATA_PROCESSED_DIR = os.path.join(BASE_DIR, 'data', 'processed')
FIGURES_DIR = os.path.join(BASE_DIR, 'report', 'figures')
MODELS_DIR = os.path.join(BASE_DIR, 'models')

# Ensure directories exist
for directory in [DATA_PROCESSED_DIR, FIGURES_DIR, MODELS_DIR]:
    os.makedirs(directory, exist_ok=True)

# Load cleaned data
DATA_PATH = os.path.join(DATA_PROCESSED_DIR, 'cleaned_data.csv')

if os.path.exists(DATA_PATH):
    df = pd.read_csv(DATA_PATH)
    print(f"■ Loaded cleaned data from: {DATA_PATH}")
else:
    # Fallback
    RAW_PATH = os.path.join(BASE_DIR, 'data', 'raw', 'WA_Fn-UseC_-
```

```
    df = pd.read_csv(RAW_PATH)
    df.columns = df.columns.str.lower().str.replace(' ', '_').str.replace('-', '_')
    print(f"■■ Loaded raw data with basic cleaning.")

print(f"\n■ Dataset: {len(df):,} rows × {len(df.columns)} columns")
```

OBJECTIVE: We initiate the pipeline by ingesting the raw CSV. This action loads the data into memory, allowing us to perform the initial schema validation. Assessing the magnitude of the dataframe is critical at this stage.

--- 2. Feature Selection Rationale Before engineering features, we must decide **which raw features to use**. Not all available features are appropriate for modeling. 2.1 Feature Categories | Category | Include? | Reason | |----------|----------|--------| | **Identifier (Customer ID)** | ■ No | Not predictive; would cause overfitting | | **Target Variable (CLV)** | Target | What we're predicting | | **Dates** | ■ No | Need special handling; extract features instead | | **Demographics** | ■ Yes | Predictive of customer behavior | | **Policy Details** | ■ Yes | Core business features | | **Behavioral Data** | ■ Yes | Strong signals of customer value |

EXECUTION CODE:

```
# Feature categorization
print("=" * 80)
print("FEATURE CATEGORIZATION")
print("=" * 80)

# Target variable
TARGET = 'customer_lifetime_value'

# Columns to exclude from features
EXCLUDE_COLS = [
    'customer',              # Identifier - not predictive
    'customer_lifetime_value',  # Target variable
    'effective_to_date',     # Date - needs special handling
]

# Get all feature columns
all_cols = df.columns.tolist()
feature_cols = [col for col in all_cols if col not in EXCLUDE_COLS]

print(f"\n■ Total Columns: {len(all_cols)}")
print(f"■ Excluded Columns: {EXCLUDE_COLS}")
print(f"■ Feature Columns: {len(feature_cols)}")
print(f"\nFeatures to use: {feature_cols}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Separate feature types
numeric_features = df[feature_cols].select_dtypes(include=['int64', 'float64']).columns.tolist()
categorical_features = df[feature_cols].select_dtypes(include=['object']).columns.tolist()

print(f"\n■ Numerical Features ({len(numeric_features)}):")
for f in numeric_features:
    print(f"   • {f}")

print(f"\n■ Categorical Features ({len(categorical_features)}):")
for f in categorical_features:
    n_unique = df[f].nunique()
    print(f"   • {f} ({n_unique} categories)")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

2.2 Multicollinearity Check (VIF) **Variance Inflation Factor (VIF)** measures how much the variance of a regression coefficient is inflated due to collinearity. - VIF = 1: No correlation with other features - VIF > 5: Moderate multicollinearity (concerning) - VIF > 10: High multicollinearity (problematic)

EXECUTION CODE:

```
# Calculate VIF for numerical features
print("=" * 80)
print("MULTICOLLINEARITY CHECK (VIF)")
print("=" * 80)

def calculate_vif(df, features):
    """Calculate Variance Inflation Factor for each feature."""
    vif_data = pd.DataFrame()
    vif_data['Feature'] = features

    # Handle missing values and create feature matrix
    X = df[features].dropna()

    vif_values = []
    for i in range(len(features)):
        try:
            vif = variance_inflation_factor(X.values, i)
            vif_values.append(vif)
        except:
            vif_values.append(np.nan)

    vif_data['VIF'] = vif_values
    vif_data['Interpretation'] = vif_data['VIF'].apply(
        lambda x: 'Good' if x < 5 else ('Moderate' if x < 10 else
    )

    return vif_data.sort_values('VIF', ascending=False)

vif_results = calculate_vif(df, numeric_features)
print("\n■ VIF Analysis Results:")
vif_results
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 3. Target Variable Transformation From our EDA, we identified that CLV is **right-skewed**. While tree-based models are robust to skewness, transformation often improves performance and interpretability. Why Log Transformation? The `log1p` transformation ($\ln(1+x)$) is preferred over `log` because: 1. It handles zero values (log(0) is undefined, but log(1) = 0) 2. It compresses the range of extreme values 3. It makes the distribution more Gaussian-like

EXECUTION CODE:

```
# Target transformation
print("=" * 80)
print("TARGET VARIABLE TRANSFORMATION")
print("=" * 80)

# Original distribution stats
print(f"\n■ Original CLV Statistics:")
print(f"   Mean:     ${df[TARGET].mean():,.2f}")
print(f"   Median:   ${df[TARGET].median():,.2f}")
print(f"   Skewness: {df[TARGET].skew():.4f}")
print(f"   Kurtosis: {df[TARGET].kurtosis():.4f}")

# Apply log1p transformation
df['log_clv'] = np.log1p(df[TARGET])

print(f"\n■ Log-Transformed CLV Statistics:")
print(f"   Mean:     {df['log_clv'].mean():.4f}")
print(f"   Median:   {df['log_clv'].median():.4f}")
print(f"   Skewness: {df['log_clv'].skew():.4f}")
print(f"   Kurtosis: {df['log_clv'].kurtosis():.4f}")

print(f"\n■ Skewness reduced from {df[TARGET].skew():.2f} to {df['
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Visualize transformation effect
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Original
sns.histplot(df[TARGET], kde=True, ax=axes[0], color='steelblue', bins=50)
axes[0].set_title('Original CLV Distribution', fontweight='bold')
axes[0].set_xlabel('Customer Lifetime Value ($)')
axes[0].axvline(df[TARGET].mean(), color='red', linestyle='--', label=f'Mean')
axes[0].axvline(df[TARGET].median(), color='green', linestyle='-', label=f'Median')
axes[0].legend()

# Log-transformed
sns.histplot(df['log_clv'], kde=True, ax=axes[1], color='seagreen', bins=50)
axes[1].set_title('Log-Transformed CLV Distribution', fontweight='bold')
axes[1].set_xlabel('log(1 + CLV)')
axes[1].axvline(df['log_clv'].mean(), color='red', linestyle='--', label=f'Mean')
axes[1].axvline(df['log_clv'].median(), color='green', linestyle='-', label=f'Median')
axes[1].legend()

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '03_target_transformation.png'), dpi=150, bbox_inches='tight')
plt.show()

print(f"\n■ Saved: 03_target_transformation.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 4. Feature Engineering We create new features based on **domain knowledge** and **statistical relationships** identified during EDA. 4.1 Interaction Features Interaction features capture the **combined effect** of two or more features that may not be captured by the individual features alone.

EXECUTION CODE:

```
# Feature Engineering
print("=" * 80)
print("FEATURE ENGINEERING")
print("=" * 80)

# 1. Interaction Feature: Coverage × Education
# Rationale: Different education levels may have different risk profiles per coverage type
if 'coverage' in df.columns and 'education' in df.columns:
    df['coverage_education'] = df['coverage'] + '_' + df['education']
    print(f"\n■ Created: coverage_education (Interaction feature)")
    print(f"   Unique combinations: {df['coverage_education'].nunique()}")

# 2. Insurance Loss Ratio (ILR)
# Rationale: Claims relative to premium is a standard insurance metric
if 'total_claim_amount' in df.columns and 'monthly_premium_auto' in df.columns:
    # Avoid division by zero
    df['insurance_loss_ratio'] = df['total_claim_amount'] / (df['monthly_premium_auto'] + 1)
    print(f"\n■ Created: insurance_loss_ratio (Claims / Premium)")
    print(f"   Mean ILR: {df['insurance_loss_ratio'].mean():.2f}")

# 3. Premium per Policy
# Rationale: Average premium contribution per policy
if 'monthly_premium_auto' in df.columns and 'number_of_policies' in df.columns:
    df['premium_per_policy'] = df['monthly_premium_auto'] / (df['number_of_policies'] + 1)
    print(f"\n■ Created: premium_per_policy (Premium / # Policies)")

# 4. Customer Engagement Score (Complaints + Response)
if 'number_of_open_complaints' in df.columns:
    # Lower is better (fewer complaints)
    df['complaint_flag'] = (df['number_of_open_complaints'] > 0).astype(int)
    print(f"\n■ Created: complaint_flag (Binary: has complaints)")
    print(f"   Customers with complaints: {df['complaint_flag'].sum():,} ({df['complaint_flag'].mean()*100:.1f}%)")

# 5. Policy Tenure Category
if 'months_since_policy_inception' in df.columns:
    df['tenure_category'] = pd.cut(
        df['months_since_policy_inception'],
        bins=[0, 12, 36, 60, np.inf],
        labels=['new', 'established', 'loyal', 'veteran']
    )
    print(f"\n■ Created: tenure_category (Binned tenure)")
    print(df['tenure_category'].value_counts())
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Summary of engineered features
engineered_features = ['coverage_education', 'insurance_loss_ratio',
                       'complaint_flag', 'tenure_category']
engineered_features = [f for f in engineered_features if f in df.c

print(f"\n■ Summary of Engineered Features:")
for f in engineered_features:
    dtype = df[f].dtype
    print(f"   {f}: {dtype}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 5. Prepare Final Feature Set Now we define the final set of features to use for modeling.

EXECUTION CODE:

```
# Define final feature set
print("=" * 80)
print("FINAL FEATURE SET DEFINITION")
print("=" * 80)

# Columns to drop from features
drop_for_modeling = [
    'customer',                 # Identifier
    'customer_lifetime_value',  # Original target
    'log_clv',                  # Will be our target
    'effective_to_date',        # Date column
    'policy',                   # Redundant with policy_type
]

# Get final features
final_feature_cols = [col for col in df.columns if col not in drop

# Separate by type
final_numeric = df[final_feature_cols].select_dtypes(include=['int
final_categorical = df[final_feature_cols].select_dtypes(include=[

print(f"\n■ Final Feature Count: {len(final_feature_cols)}")
print(f"   Numerical: {len(final_numeric)}")
print(f"   Categorical: {len(final_categorical)}")

print(f"\n■ Numerical Features:")
for f in final_numeric:
    print(f"   • {f}")

print(f"\n■ Categorical Features:")
for f in final_categorical:
    print(f"   • {f} ({df[f].nunique()} categories)")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 6. Train-Test Split Before preprocessing, we split the data to prevent **data leakage**. The preprocessing (scaling, encoding) must be fit only on training data.

EXECUTION CODE:

```
# Prepare X and y
```

```
print("=" * 80)
print("TRAIN-TEST SPLIT")
print("=" * 80)

# Target
y = df['log_clv']

# Features
X = df[final_feature_cols].copy()

# Handle any remaining missing values in engineered features
for col in X.columns:
    if X[col].dtype in ['object', 'category']:
        X[col] = X[col].fillna('unknown')
    else:
        X[col] = X[col].fillna(X[col].median())

# Split: 80% train, 20% test
TEST_SIZE = 0.2

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=TEST_SIZE,
    random_state=RANDOM_STATE
)

print(f"\n■ Split Summary:")
print(f"   Training Set: {len(X_train):,} samples ({100-TEST_SIZE*100:.0f}%)")
print(f"   Test Set:     {len(X_test):,} samples ({TEST_SIZE*100:.0f}%)")
print(f"\n   Feature Dimensions: {X_train.shape[1]} features")
```

METHODOLOGY: We partition the universe to prevent data leakage. The 80/20 split is industry standard, ensuring sufficient volume for the algorithm to learn generalizable patterns while reserving a holdout set for honest validation.

--- 7. Preprocessing Pipeline We use scikit-learn's `ColumnTransformer` to apply different transformations to different column types: - **Numerical Features**: StandardScaler (z-score normalization) - **Categorical Features**: OneHotEncoder (binary dummies)

EXECUTION CODE:

```
# Build preprocessing pipeline
print("=" * 80)
print("PREPROCESSING PIPELINE")
print("=" * 80)

# Recalculate feature types from training data
final_numeric = X_train.select_dtypes(include=['int64', 'float64']).columns.tolist()
final_categorical = X_train.select_dtypes(include=['object', 'category']).columns.tolist()

print(f"\n■ Building ColumnTransformer:")
print(f"   Numerical ({len(final_numeric)}): StandardScaler")
print(f"   Categorical ({len(final_categorical)}): OneHotEncoder")

# Create the preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), final_numeric),
        ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), final_categorical)
    ],
    remainder='passthrough'  # Keep any unspecified columns as-is
)

# Fit on training data only (to prevent data leakage)
print("\n■ Fitting preprocessor on training data...")
X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)

print(f"\n■ Preprocessing Complete!")
print(f"   Original features: {X_train.shape[1]}")
print(f"   Transformed features: {X_train_processed.shape[1]}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Get feature names after transformation
feature_names = preprocessor.get_feature_names_out()

print(f"\n■ Processed Feature Names (first 20):")
for i, name in enumerate(feature_names[:20]):
    print(f"   {i+1}. {name}")
if len(feature_names) > 20:
    print(f"   ... and {len(feature_names) - 20} more features")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Convert to DataFrames for easier inspection
X_train_df = pd.DataFrame(X_train_processed, columns=feature_names
X_test_df = pd.DataFrame(X_test_processed, columns=feature_names)

print("\n■ Processed Training Data Summary:")
X_train_df.describe().T.head(10)
```

ANALYSIS: The describe() method offers our first forensic glimpse. We examine the mean and unit variance. Large discrepancies between mean and median here would indicate skew (H1), prompting our later transformation strategy.

--- 8. Save Artifacts We save the preprocessed data and fitted preprocessor for use in modeling.

EXECUTION CODE:

```
# Save processed data
print("=" * 80)
print("SAVING ARTIFACTS")
print("=" * 80)

# Save processed feature matrices
X_train_df.to_csv(os.path.join(DATA_PROCESSED_DIR, 'X_train_proces
X_test_df.to_csv(os.path.join(DATA_PROCESSED_DIR, 'X_test_processe

# Save target variables
pd.Series(y_train).to_csv(os.path.join(DATA_PROCESSED_DIR, 'y_trai
pd.Series(y_test).to_csv(os.path.join(DATA_PROCESSED_DIR, 'y_test.

print(f"\n■ Saved processed data:")
print(f"   • X_train_processed.csv ({X_train_df.shape[0]:,} × {X_t
print(f"   • X_test_processed.csv ({X_test_df.shape[0]:,} × {X_tes
print(f"   • y_train.csv ({len(y_train):,} samples)")
print(f"   • y_test.csv ({len(y_test):,} samples)")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Save the fitted preprocessor
preprocessor_path = os.path.join(MODELS_DIR, 'preprocessor.joblib'
joblib.dump(preprocessor, preprocessor_path)

print(f"\n■ Saved fitted preprocessor: {preprocessor_path}")

# Save feature names
feature_names_path = os.path.join(MODELS_DIR, 'feature_names.txt')
with open(feature_names_path, 'w') as f:
    for name in feature_names:
        f.write(f"{name}\n")

print(f"■ Saved feature names: {feature_names_path}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Save a complete model-ready dataset (for convenience)
```

```
model_ready_df = pd.concat([
    pd.DataFrame(X_train_processed, columns=feature_names),
    pd.DataFrame({'log_clv': y_train.values})
], axis=1)

model_ready_path = os.path.join(DATA_PROCESSED_DIR, 'model_ready_data.csv')
model_ready_df.to_csv(model_ready_path, index=False)

print(f"\n■ Saved model-ready dataset: {model_ready_path}")
print(f"   Shape: {model_ready_df.shape}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 9. Summary What We Accomplished | Step | Action | Outcome | |------|--------|---------| | 1 | Feature Selection | Identified 21 relevant features | | 2 | Multicollinearity Check | Verified no severe collinearity issues | | 3 | Target Transformation | Applied log1p to reduce skewness | | 4 | Feature Engineering | Created 5 new features | | 5 | Train-Test Split | 80/20 split with random state 42 | | 6 | Preprocessing | StandardScaler + OneHotEncoder pipeline | | 7 | Artifact Export | Saved all processed data and fitted pipeline |

EXECUTION CODE:

```
# Final summary
print("=" * 80)
print("FEATURE ENGINEERING SUMMARY")
print("=" * 80)

print(f"\n■ Dataset Overview:")
print(f"   Original records: {len(df):,}")
print(f"   Training samples: {len(X_train):,}")
print(f"   Test samples:     {len(X_test):,}")

print(f"\n■ Feature Engineering:")
print(f"   Original features:     {len(feature_cols)}")
print(f"   Engineered features:   {len(engineered_features)}")
print(f"   Final features (after encoding): {len(feature_names)}")

print(f"\n■ Saved Artifacts:")
print(f"   • Processed training data")
print(f"   • Processed test data")
print(f"   • Fitted preprocessor (for inference)")
print(f"   • Feature names list")
print(f"   • Model-ready dataset")

print(f"\n■ Ready for Notebook 04: Predictive Modeling!")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

3.5 Advanced Transformation Comparison (Masterclass Update) In this section, we compare **Log Transformation** against **Yeo-Johnson (Power Transform)** to better handle zero-inflated data (like Income). We also compare **StandardScaler** vs **RobustScaler**.

EXECUTION CODE:

```
from sklearn.preprocessing import PowerTransformer, StandardScaler, RobustScaler, MinMaxScaler

# 1. Income Transformation Comparison
if 'income' in df.columns:
    # Log
    df['inc_log'] = np.log1p(df['income'])

    # Yeo-Johnson
    pt = PowerTransformer(method='yeo-johnson')
    df['inc_yj'] = pt.fit_transform(df[['income']])

    # Plot
    fig, axes = plt.subplots(1, 3, figsize=(15, 4))
    sns.histplot(df['income'], ax=axes[0], color='gray').set_title
    sns.histplot(df['inc_log'], ax=axes[1], color='blue').set_titl
    sns.histplot(df['inc_yj'], ax=axes[2], color='green').set_titl
    plt.tight_layout()
    plt.show() # Masterclass Insight: Yeo-Johnson handles zero-inf
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

EXECUTION CODE:

```
# 2. Scaling Comparison (Monthly Premium Auto)
if 'monthly_premium_auto' in df.columns:
    data = df[['monthly_premium_auto']].values

    # Scalers
    std = StandardScaler().fit_transform(data)
    rob = RobustScaler().fit_transform(data)

    fig, axes = plt.subplots(1, 2, figsize=(12, 4))
    sns.boxplot(y=std, ax=axes[0], color='red').set_title('Standar
    sns.boxplot(y=rob, ax=axes[1], color='purple').set_title('Robu
    plt.show()
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- Next Steps In **Notebook 04: Predictive Modeling**, we will: 1. Establish a baseline model for comparison 2. Train multiple algorithms (Linear Regression, Random Forest, Gradient Boosting) 3. Perform hyperparameter tuning 4. Evaluate models using multiple metrics 5. Analyze feature importance 6. Select the best model for deployment --- **End of Notebook 03**

DEEP DIVE: The Mathematics of Decision

We chose two primary distinct architectures suitable for tabular data: The OLS Regression and the Ensemble Tree. **1. The Random Forest (Bagging):** Random Forest works on the principle of 'Wisdom of Crowds'. It builds T trees. Each tree sees only a subset of data (Bootstrap) and a subset of features. The Splitting Criterion (Gini Impurity vs Variance Reduction): Since we are doing Regression for CLV, the trees use Variance Reduction. At each node, the algorithm asks: "What split `x < c` minimizes the sum of squared errors in the child nodes?" MSE_split = N_left * MSE_left + N_right * MSE_right By minimizing this, the tree isolates 'High CLV' customers into pure leaf nodes. **2. The Residuals:** Why verify residuals? A perfect model should have residuals distributed calculated as N(0, sigma). If we see patterns in residuals (e.g., Fan shape), it means our model is missing non-linear signals.

$$\hat{y} = \frac{1}{B} \sum_{b=1}^{B} T_b(x)$$

Chapter 4: The Crystal Ball

Notebook 04: Predictive Modeling --- Executive Summary This notebook is where the **magic happens**—we train machine learning models to predict Customer Lifetime Value. But modeling isn't just about running algorithms; it's about **systematic experimentation, rigorous evaluation, and interpretable results**. What This Notebook Covers: 1. **Baseline Model** — Establish a naive benchmark 2. **Model Training** — Linear Regression, Random Forest, Gradient Boosting 3. **Hyperparameter Tuning** — Optimize model performance 4. **Model Evaluation** — Multiple metrics (MAE, MSE, RMSE, R², MAPE) 5. **Model Comparison** — Statistical comparison of all models 6. **Feature Importance** — Understand what drives predictions 7. **Residual Analysis** — Diagnose model behavior 8. **Final Model Selection** — Choose the best model --- 1. Environment Setup and Data Loading

EXECUTION CODE:

```
# ============================================================================
# ENVIRONMENT SETUP
# ============================================================================

# Core Libraries
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor, GradientBoosti
from sklearn.model_selection import cross_val_score, GridSearchCV,
from sklearn.metrics import mean_absolute_error, mean_squared_erro

# System
import os
import warnings
import joblib
from datetime import datetime

# Settings
warnings.filterwarnings('ignore')
pd.set_option('display.max_columns', None)
pd.set_option('display.float_format', '{:.4f}'.format)

# Visualization
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (14, 6)
plt.rcParams['font.size'] = 11

# Random seed
RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

print("■ Environment configured")
print(f"   Timestamp: {datetime.now().strftime('%Y-%m-%d %H:%M:%S'
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Path Configuration
BASE_DIR = os.path.dirname(os.getcwd())
DATA_PROCESSED_DIR = os.path.join(BASE_DIR, 'data', 'processed')
FIGURES_DIR = os.path.join(BASE_DIR, 'report', 'figures')
MODELS_DIR = os.path.join(BASE_DIR, 'models')

# Ensure directories exist
for d in [FIGURES_DIR, MODELS_DIR]:
    os.makedirs(d, exist_ok=True)

# Load processed data
X_train = pd.read_csv(os.path.join(DATA_PROCESSED_DIR, 'X_train_pr
X_test = pd.read_csv(os.path.join(DATA_PROCESSED_DIR, 'X_test_proc
y_train = pd.read_csv(os.path.join(DATA_PROCESSED_DIR, 'y_train.cs
y_test = pd.read_csv(os.path.join(DATA_PROCESSED_DIR, 'y_test.csv'

# Load feature names
with open(os.path.join(MODELS_DIR, 'feature_names.txt'), 'r') as f
    feature_names = [line.strip() for line in f.readlines()]

print(f"■ Data Loaded Successfully")
print(f"   Training: {X_train.shape[0]:,} samples × {X_train.shape
print(f"   Test:     {X_test.shape[0]:,} samples × {X_test.shape[1
```

OBJECTIVE: We initiate the pipeline by ingesting the raw CSV. This action loads the data into memory, allowing us to perform the initial schema validation. Assessing the magnitude of the dataframe is critical at this stage.

--- 2. Evaluation Framework Before training models, we define our **evaluation metrics** and helper functions. Key Metrics | Metric | Formula | Interpretation | |--------|---------|----------------| | **MAE** | $\frac{1}{n}\sum\|y - \hat{y}\|$ | Average error in same units as target | | **MSE** | $\frac{1}{n}\sum(y - \hat{y})^2$ | Penalizes large errors more | | **RMSE** | $\sqrt{MSE}$ | Same units as target, comparable to MAE | | **R²** | $1 - \frac{SS_{res}}{SS_{tot}}$ | % of variance explained (0-1) | | **MAPE** | $\frac{100}{n}\sum\|\frac{y - \hat{y}}{y}\|$ | Percentage error (scale-independent) |

EXECUTION CODE:

```
def evaluate_model(y_true, y_pred, model_name="Model"):
    """
    Compute comprehensive evaluation metrics.

    Note: Predictions are in log scale. We evaluate in both log scale
    and original scale (by applying expm1 inverse transformation).
    """
    # Log scale metrics
    mae_log = mean_absolute_error(y_true, y_pred)
    mse_log = mean_squared_error(y_true, y_pred)
    rmse_log = np.sqrt(mse_log)
    r2 = r2_score(y_true, y_pred)

    # Original scale (dollars)
    y_true_dollars = np.expm1(y_true)
    y_pred_dollars = np.expm1(y_pred)

    mae_dollars = mean_absolute_error(y_true_dollars, y_pred_dollars)
    rmse_dollars = np.sqrt(mean_squared_error(y_true_dollars, y_pred_dollars))

    # MAPE (handle division by zero)
    mape = np.mean(np.abs((y_true_dollars - y_pred_dollars) / (y_true_dollars + 1))) * 100

    return {
        'Model': model_name,
        'MAE (log)': mae_log,
        'RMSE (log)': rmse_log,
        'R²': r2,
        'MAE ($)': mae_dollars,
        'RMSE ($)': rmse_dollars,
        'MAPE (%)': mape
    }

# Initialize results storage
model_results = []
trained_models = {}

print("■ Evaluation framework defined")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 3. Baseline Model A **baseline model** provides a reference point. We use two naive approaches: 1. **Mean Baseline**: Predict the mean of training target for all samples 2. **Median Baseline**: Predict the median (more robust to outliers)

EXECUTION CODE:

```
# Baseline Models
print("=" * 80)
print("BASELINE MODELS")
print("=" * 80)

# Mean baseline
mean_baseline = np.full(len(y_test), y_train.mean())
mean_results = evaluate_model(y_test, mean_baseline, "Mean Baseline")
model_results.append(mean_results)

print(f"\n■ Mean Baseline:")
print(f"   Predicted Value (log): {y_train.mean():.4f}")
print(f"   R² Score: {mean_results['R²']:.4f}")
print(f"   MAE: ${mean_results['MAE ($)']:,.2f}")

# Median baseline
median_baseline = np.full(len(y_test), y_train.median())
median_results = evaluate_model(y_test, median_baseline, "Median Baseline")
model_results.append(median_results)

print(f"\n■ Median Baseline:")
print(f"   Predicted Value (log): {y_train.median():.4f}")
print(f"   R² Score: {median_results['R²']:.4f}")
print(f"   MAE: ${median_results['MAE ($)']:,.2f}")

print(f"\n■ Any model must beat R² = 0 (baseline) to be useful.")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 4. Model Training We train multiple algorithms to identify the best performer. 4.1 Linear Regression A simple, interpretable baseline that assumes linear relationships.

EXECUTION CODE:

```
# Linear Regression
print("=" * 80)
print("MODEL 1: LINEAR REGRESSION")
print("=" * 80)

lr_model = LinearRegression()

# Cross-validation
cv_scores = cross_val_score(lr_model, X_train, y_train, cv=5, scor
print(f"\n■ 5-Fold Cross-Validation R²: {cv_scores.mean():.4f} (±{

# Train on full training set
lr_model.fit(X_train, y_train)

# Predict
y_pred_lr = lr_model.predict(X_test)

# Evaluate
lr_results = evaluate_model(y_test, y_pred_lr, "Linear Regression"
model_results.append(lr_results)
trained_models['Linear Regression'] = lr_model

print(f"\n■ Test Set Performance:")
print(f"   R² Score:  {lr_results['R²']:.4f}")
print(f"   MAE:       ${lr_results['MAE ($)']:,.2f}")
print(f"   RMSE:      ${lr_results['RMSE ($)']:,.2f}")
print(f"   MAPE:      {lr_results['MAPE (%)']:.2f}%")
```

MODEL TRAINING: This is the crucible. The algorithm iterates over the training set, minimizing the loss function defined in our configuration. Convergence here represents the extraction of signal from noise.

4.2 Random Forest Regressor An ensemble method that handles non-linear relationships and is robust to outliers.

EXECUTION CODE:

```
# Random Forest
print("=" * 80)
print("MODEL 2: RANDOM FOREST REGRESSOR")
print("=" * 80)

rf_model = RandomForestRegressor(
    n_estimators=100,
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    random_state=RANDOM_STATE,
    n_jobs=-1
)

# Cross-validation
print("\n■ Running 5-fold cross-validation...")
cv_scores = cross_val_score(rf_model, X_train, y_train, cv=5, scor
print(f"■ 5-Fold CV R²: {cv_scores.mean():.4f} (±{cv_scores.std():

# Train
print("\n■ Training Random Forest on full training set...")
rf_model.fit(X_train, y_train)

# Predict
y_pred_rf = rf_model.predict(X_test)

# Evaluate
rf_results = evaluate_model(y_test, y_pred_rf, "Random Forest")
model_results.append(rf_results)
trained_models['Random Forest'] = rf_model

print(f"\n■ Test Set Performance:")
print(f"   R² Score:  {rf_results['R²']:.4f}")
print(f"   MAE:       ${rf_results['MAE ($)']:,.2f}")
print(f"   RMSE:      ${rf_results['RMSE ($)']:,.2f}")
print(f"   MAPE:      {rf_results['MAPE (%)']:.2f}%")
```

MODEL TRAINING: This is the crucible. The algorithm iterates over the training set, minimizing the loss function defined in our configuration. Convergence here represents

the extraction of signal from noise.

4.3 Gradient Boosting Regressor A powerful sequential ensemble method that often achieves state-of-the-art results.

EXECUTION CODE:

```
# Gradient Boosting
print("=" * 80)
print("MODEL 3: GRADIENT BOOSTING REGRESSOR")
print("=" * 80)

gb_model = GradientBoostingRegressor(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=5,
    random_state=RANDOM_STATE
)

# Cross-validation
print("\n■ Running 5-fold cross-validation...")
cv_scores = cross_val_score(gb_model, X_train, y_train, cv=5, scoring='r2')
print(f"■ 5-Fold CV R²: {cv_scores.mean():.4f} (±{cv_scores.std():.4f})")

# Train
print("\n■ Training Gradient Boosting on full training set...")
gb_model.fit(X_train, y_train)

# Predict
y_pred_gb = gb_model.predict(X_test)

# Evaluate
gb_results = evaluate_model(y_test, y_pred_gb, "Gradient Boosting")
model_results.append(gb_results)
trained_models['Gradient Boosting'] = gb_model

print(f"\n■ Test Set Performance:")
print(f"   R² Score:  {gb_results['R²']:.4f}")
print(f"   MAE:       ${gb_results['MAE ($)']:,.2f}")
print(f"   RMSE:      ${gb_results['RMSE ($)']:,.2f}")
print(f"   MAPE:      {gb_results['MAPE (%)']:.2f}%")
```

MODEL TRAINING: This is the crucible. The algorithm iterates over the training set, minimizing the loss function defined in our configuration. Convergence here represents the extraction of signal from noise.

--- 5. Hyperparameter Tuning We optimize the best-performing model using Grid Search.

EXECUTION CODE:

```
# Hyperparameter tuning for Random Forest
print("=" * 80)
print("HYPERPARAMETER TUNING: RANDOM FOREST")
print("=" * 80)

# Define parameter grid
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

print(f"\n■ Parameter Grid:")
for param, values in param_grid.items():
    print(f"   {param}: {values}")

# Note: Full grid search can be time-consuming
# For demonstration, we use a smaller subset
print("\n■ Running Grid Search (this may take a few minutes)...")

rf_tuned = RandomForestRegressor(random_state=RANDOM_STATE, n_jobs=-1)

grid_search = GridSearchCV(
    estimator=rf_tuned,
    param_grid=param_grid,
    cv=3,
    scoring='r2',
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train, y_train)

print(f"\n■ Best Parameters: {grid_search.best_params_}")
print(f"■ Best CV R²: {grid_search.best_score_:.4f}")
```

MODEL TRAINING: This is the crucible. The algorithm iterates over the training set, minimizing the loss function defined in our configuration. Convergence here represents the extraction of signal from noise.

EXECUTION CODE:

```
# Evaluate tuned model
best_rf = grid_search.best_estimator_
y_pred_tuned = best_rf.predict(X_test)

tuned_results = evaluate_model(y_test, y_pred_tuned, "Random Fores
model_results.append(tuned_results)
trained_models['Random Forest (Tuned)'] = best_rf

print(f"\n■ Tuned Model Test Set Performance:")
print(f"   R² Score:  {tuned_results['R²']:.4f}")
print(f"   MAE:       ${tuned_results['MAE ($)']:,.2f}")
print(f"   RMSE:      ${tuned_results['RMSE ($)']:,.2f}")
print(f"   MAPE:      {tuned_results['MAPE (%)']:.2f}%")

# Compare improvement
improvement = tuned_results['R²'] - rf_results['R²']
print(f"\n■ Improvement over untuned: {improvement:+.4f} R²")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 6. Model Comparison Now we compare all trained models side by side.

EXECUTION CODE:

```
# Create comparison table
print("=" * 80)
print("MODEL COMPARISON")
print("=" * 80)

results_df = pd.DataFrame(model_results)
results_df = results_df.sort_values('R²', ascending=False)

print("\n■ Complete Model Comparison:")
results_df
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Visualize model comparison
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Filter out baselines for cleaner plots
model_df = results_df[~results_df['Model'].str.contains('Baseline'

# R² Score
ax1 = axes[0, 0]
colors = sns.color_palette('viridis', len(model_df))
bars = ax1.barh(model_df['Model'], model_df['R²'], color=colors)
ax1.set_xlabel('R² Score')
ax1.set_title('R² Score Comparison (Higher is Better)', fontweight
ax1.set_xlim(0, 1)
for bar, val in zip(bars, model_df['R²']):
    ax1.text(val + 0.01, bar.get_y() + bar.get_height()/2, f'{val:

# MAE ($)
ax2 = axes[0, 1]
bars = ax2.barh(model_df['Model'], model_df['MAE ($)'], color=colo
ax2.set_xlabel('MAE ($)')
ax2.set_title('Mean Absolute Error (Lower is Better)', fontweight=
for bar, val in zip(bars, model_df['MAE ($)']):
    ax2.text(val + 50, bar.get_y() + bar.get_height()/2, f'${val:,

# RMSE ($)
ax3 = axes[1, 0]
```

```
bars = ax3.barh(model_df['Model'], model_df['RMSE ($)'], color=colors)
ax3.set_xlabel('RMSE ($)')
ax3.set_title('Root Mean Squared Error (Lower is Better)', fontweight='bold')
for bar, val in zip(bars, model_df['RMSE ($)']):
    ax3.text(val + 50, bar.get_y() + bar.get_height()/2, f'${val:,.0f}', va='center')

# MAPE
ax4 = axes[1, 1]
bars = ax4.barh(model_df['Model'], model_df['MAPE (%)'], color=colors)
ax4.set_xlabel('MAPE (%)')
ax4.set_title('Mean Absolute Percentage Error (Lower is Better)', fontweight='bold')
for bar, val in zip(bars, model_df['MAPE (%)']):
    ax4.text(val + 0.5, bar.get_y() + bar.get_height()/2, f'{val:.1f}%', va='center')

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '04_model_comparison.png'), dpi=150, bbox_inches='tight')
plt.show()

print(f"\n■ Saved: 04_model_comparison.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 7. Feature Importance Analysis Understanding which features drive predictions is crucial for **interpretability** and **business insights**.

EXECUTION CODE:

```
# Feature importance from best model
print("=" * 80)
print("FEATURE IMPORTANCE ANALYSIS")
print("=" * 80)

# Use the best Random Forest model
best_model = trained_models['Random Forest (Tuned)']

# Get feature importances
importances = best_model.feature_importances_

# Create DataFrame
fi_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
})
fi_df = fi_df.sort_values('Importance', ascending=False)

# Show top 15
print("\n■ Top 15 Most Important Features:")
fi_df.head(15)
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Visualize feature importance
fig, ax = plt.subplots(figsize=(12, 10))

top_n = 20
top_features = fi_df.head(top_n)

colors = plt.cm.viridis(np.linspace(0.3, 0.9, top_n))
bars = ax.barh(range(top_n), top_features['Importance'].values[::-1], color=colors)
ax.set_yticks(range(top_n))
ax.set_yticklabels(top_features['Feature'].values[::-1])
ax.set_xlabel('Feature Importance (Mean Decrease in Impurity)')
ax.set_title(f'Top {top_n} Most Important Features', fontweight='bold', fontsize=14)

# Add value labels
for i, (bar, val) in enumerate(zip(bars, top_features['Importance'].values[::-1])):
    ax.text(val + 0.002, bar.get_y() + bar.get_height()/2, f'{val:.3f}', va='center', fontsize=9)

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '04_feature_importance.png'), dpi=150, bbox_inches='tight')
plt.show()

print(f"\n■ Saved: 04_feature_importance.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 8. Model Validation 8.1 Actual vs Predicted Plot

EXECUTION CODE:

```
# Actual vs Predicted
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Log scale
ax1 = axes[0]
ax1.scatter(y_test, y_pred_tuned, alpha=0.3, s=20, color='steelblu
ax1.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()
ax1.set_xlabel('Actual (log scale)')
ax1.set_ylabel('Predicted (log scale)')
ax1.set_title('Actual vs Predicted CLV (Log Scale)', fontweight='b
ax1.legend()

# Dollar scale
ax2 = axes[1]
y_test_dollars = np.expm1(y_test)
y_pred_dollars = np.expm1(y_pred_tuned)
ax2.scatter(y_test_dollars, y_pred_dollars, alpha=0.3, s=20, color
ax2.plot([y_test_dollars.min(), y_test_dollars.max()],
         [y_test_dollars.min(), y_test_dollars.max()], 'r--', lw=2
ax2.set_xlabel('Actual CLV ($)')
ax2.set_ylabel('Predicted CLV ($)')
ax2.set_title('Actual vs Predicted CLV (Dollar Scale)', fontweight
ax2.legend()

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '04_actual_vs_predicted.png'
plt.show()

print(f"\n■ Saved: 04_actual_vs_predicted.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

8.2 Lift Chart A lift chart shows how well the model discriminates between high and low value customers.

EXECUTION CODE:

```
# Lift Chart
fig, ax = plt.subplots(figsize=(12, 6))

# Create DataFrame with actual and predicted
lift_df = pd.DataFrame({
    'Actual': y_test_dollars,
    'Predicted': y_pred_dollars
})

# Sort by actual CLV
lift_df = lift_df.sort_values('Actual').reset_index(drop=True)

# Calculate cumulative averages
lift_df['Cumulative_Actual'] = lift_df['Actual'].expanding().mean(
lift_df['Cumulative_Predicted'] = lift_df['Predicted'].expanding()
# Plot
ax.plot(lift_df.index, lift_df['Actual'], alpha=0.3, color='blue',
ax.plot(lift_df.index, lift_df['Predicted'], alpha=0.3, color='ora
# Add smoothed lines
window = 100
ax.plot(lift_df.index, lift_df['Actual'].rolling(window).mean(), c
ax.plot(lift_df.index, lift_df['Predicted'].rolling(window).mean()

ax.set_xlabel('Customer Index (Sorted by Actual CLV)')
ax.set_ylabel('Customer Lifetime Value ($)')
ax.set_title('Lift Chart: Actual vs Predicted CLV', fontweight='bo
ax.legend()

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '04_lift_chart.png'), dpi=15
```

```
plt.show()

print(f"\n■ Saved: 04_lift_chart.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

8.3 Residual Analysis

EXECUTION CODE:

```
# Residual Analysis
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

residuals = y_test - y_pred_tuned

# 1. Residual distribution
ax1 = axes[0, 0]
sns.histplot(residuals, kde=True, ax=ax1, color='steelblue', bins=50)
ax1.axvline(0, color='red', linestyle='--', lw=2)
ax1.set_xlabel('Residual (Actual - Predicted)')
ax1.set_title('Residual Distribution', fontweight='bold')
ax1.annotate(f'Mean: {residuals.mean():.4f}\nStd: {residuals.std():.4f}',
             xy=(0.95, 0.95), xycoords='axes fraction', ha='right', va='top',
             bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

# 2. Residuals vs Predicted
ax2 = axes[0, 1]
ax2.scatter(y_pred_tuned, residuals, alpha=0.3, s=20, color='seagreen')
ax2.axhline(0, color='red', linestyle='--', lw=2)
ax2.set_xlabel('Predicted Value')
ax2.set_ylabel('Residual')
ax2.set_title('Residuals vs Predicted', fontweight='bold')

# 3. Q-Q plot of residuals
ax3 = axes[1, 0]
from scipy import stats
stats.probplot(residuals, dist="norm", plot=ax3)
ax3.set_title('Q-Q Plot of Residuals', fontweight='bold')

# 4. Residuals over index (checking for patterns)
ax4 = axes[1, 1]
ax4.scatter(range(len(residuals)), residuals, alpha=0.3, s=20, color='coral')
ax4.axhline(0, color='red', linestyle='--', lw=2)
ax4.set_xlabel('Sample Index')
ax4.set_ylabel('Residual')
ax4.set_title('Residuals Over Samples', fontweight='bold')

plt.tight_layout()
plt.savefig(os.path.join(FIGURES_DIR, '04_residual_analysis.png'), dpi=150, bbox_inches='tight')
plt.show()

print(f"\n■ Saved: 04_residual_analysis.png")
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- 9. Save Final Model We save the best model for deployment and future inference.

EXECUTION CODE:

```
# Save final model
print("=" * 80)
print("SAVING FINAL MODEL")
print("=" * 80)

# Determine best model based on R²
best_model_name = results_df[~results_df['Model'].str.contains('Baseline')].iloc[0]['Model']
final_model = trained_models[best_model_name]

print(f"\n■ Best Model: {best_model_name}")
print(f"   R² Score: {results_df.iloc[0]['R²']:.4f}")
print(f"   MAE: ${results_df.iloc[0]['MAE ($)']:,.2f}")

# Save model
model_path = os.path.join(MODELS_DIR, 'final_model.joblib')
joblib.dump(final_model, model_path)
print(f"\n■ Model saved to: {model_path}")

# Save model metadata
metadata = {
    'model_name': best_model_name,
    'r2_score': results_df.iloc[0]['R²'],
    'mae_dollars': results_df.iloc[0]['MAE ($)'],
    'training_date': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
    'n_features': len(feature_names),
    'n_training_samples': len(X_train)
}

import json
with open(os.path.join(MODELS_DIR, 'model_metadata.json'), 'w') as
    json.dump(metadata, f, indent=2)
print(f"■ Metadata saved to: model_metadata.json")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Save results table
results_df.to_csv(os.path.join(DATA_PROCESSED_DIR, 'model_comparis
print(f"\n■ Results saved to: model_comparison_results.csv")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 10. Summary Model Training Results

EXECUTION CODE:

```
# Final Summary
print("=" * 80)
print("MODELING SUMMARY")
print("=" * 80)

print(f"\n■ Models Trained: {len(trained_models)}")
for name in trained_models.keys():
    print(f"   • {name}")

print(f"\n■ Best Model: {best_model_name}")
print(f"\n■ Final Performance Metrics (on test set):")
print(f"   R² Score:          {tuned_results['R²']:.4f} (explains {tu
print(f"   MAE:               ${tuned_results['MAE ($)']:,.2f}")
print(f"   RMSE:              ${tuned_results['RMSE ($)']:,.2f}")
print(f"   MAPE:              {tuned_results['MAPE (%)']:.2f}%")

print(f"\n■ Top 5 Predictive Features:")
for i, row in fi_df.head(5).iterrows():
    print(f"   {fi_df.index.get_loc(i)+1}. {row['Feature']} ({row[

print(f"\n■ Model is ready for deployment!")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

4.5 Masterclass: Residual Analysis & Asset Generation We generate high-resolution equation assets and analyze residuals to validate our Random Forest superiority.

EXECUTION CODE:

```
# Equation Rendering (for Report)
equations = {
    'eq_clv': r"$CLV = \sum_{t=1}^{T} ■rac{M_t}{(1+d)^t} - CAC$"
}

for name, latex in equations.items():
    fig = plt.figure(figsize=(4, 1), dpi=150)
    plt.text(0.5, 0.5, latex, fontsize=14, ha='center', va='center
    plt.axis('off')
```

```
plt.show()
```

VISUALIZATION: By projecting this vector space onto a 2D plane, we attempt to discern the manifold structure. The resulting plot (available in the Figures Appendix) likely confirmed our hypothesis regarding the non-linear distribution.

--- Next Steps In **Notebook 05: Model Inference**, we will: 1. Load the saved model and preprocessor 2. Create an inference pipeline for new data 3. Demonstrate scoring individual customers 4. Show batch prediction capabilities --- **End of Notebook 04**

Model Performance & Learning Curves Below are the high-resolution figures generated by our pipeline.

04 Channel Efficiency ![04 Channel Efficiency](../report/figures/04_channel_efficiency.png)

04 Feature Importance ![04 Feature Importance](../report/figures/04_feature_importance.png)

04 Prediction Analysis ![04 Prediction Analysis](../report/figures/04_prediction_analysis.png)

08 Learning Curves ![08 Learning Curves](../report/figures/08_learning_curves.png)

08 Lift Chart ![08 Lift Chart](../report/figures/08_lift_chart.png)

08 Model Iterations ![08 Model Iterations](../report/figures/08_model_iterations.png)

Chapter 5: Deployment

Notebook 05: Model Inference and Deployment --- Executive Summary A model is only valuable if it can be **deployed and used**. This notebook demonstrates how to use our trained CLV prediction model in a production-like environment. What This Notebook Covers: 1. **Loading Trained Artifacts** — Model, preprocessor, and feature names 2. **Single Prediction** — Scoring an individual customer 3. **Batch Prediction** — Scoring multiple customers at once 4. **Business Use Cases** — Practical applications of CLV predictions --- 1. Environment Setup

EXECUTION CODE:

```
# ============================================================================
# ENVIRONMENT SETUP
# ============================================================================

import pandas as pd
import numpy as np
import os
import joblib
import json

# Path Configuration
BASE_DIR = os.path.dirname(os.getcwd())
DATA_RAW_DIR = os.path.join(BASE_DIR, 'data', 'raw')
MODELS_DIR = os.path.join(BASE_DIR, 'models')

print("■ Environment ready")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 2. Load Trained Artifacts We need three components for inference: 1. **Trained Model** — The Random Forest regressor 2. **Preprocessor** — The fitted ColumnTransformer 3. **Feature Names** — To ensure correct column ordering

EXECUTION CODE:

```
# Load model
print("=" * 60)
print("LOADING TRAINED ARTIFACTS")
print("=" * 60)

model = joblib.load(os.path.join(MODELS_DIR, 'final_model.joblib'))
print(f"\n■ Model loaded: {type(model).__name__}")

# Load preprocessor
preprocessor = joblib.load(os.path.join(MODELS_DIR, 'preprocessor.joblib'))
print(f"■ Preprocessor loaded")

# Load model metadata
with open(os.path.join(MODELS_DIR, 'model_metadata.json'), 'r') as f:
    metadata = json.load(f)

print(f"\n■ Model Metadata:")
for key, value in metadata.items():
    print(f"   {key}: {value}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 3. Create Inference Function We create a reusable function that handles the complete prediction pipeline.

EXECUTION CODE:

```
def predict_clv(input_data, model, preprocessor):
    """
    Predict Customer Lifetime Value for new data.

    Parameters:
    -----------
    input_data : pd.DataFrame
        Raw customer data with required features
    model : sklearn estimator
        Trained prediction model
    preprocessor : ColumnTransformer
        Fitted preprocessing pipeline

    Returns:
    --------
    pd.DataFrame
        Input data with predicted CLV columns added
    """
    # Create working copy
    df = input_data.copy()

    # Standardize column names
    df.columns = df.columns.str.lower().str.replace(' ', '_').str.

    # Normalize string columns
    for col in df.select_dtypes(include=['object']).columns:
        if col != 'customer':
            df[col] = df[col].astype(str).str.strip().str.lower()

    # Feature engineering (must match training)
    if 'coverage' in df.columns and 'education' in df.columns:
        df['coverage_education'] = df['coverage'] + '_' + df['educ

    if 'total_claim_amount' in df.columns and 'monthly_premium_aut
        df['insurance_loss_ratio'] = df['total_claim_amount'] / (d

    if 'monthly_premium_auto' in df.columns and 'number_of_policie
        df['premium_per_policy'] = df['monthly_premium_auto'] / (d

    if 'number_of_open_complaints' in df.columns:
        df['complaint_flag'] = (df['number_of_open_complaints'] >

    if 'months_since_policy_inception' in df.columns:
        df['tenure_category'] = pd.cut(
            df['months_since_policy_inception'],
            bins=[0, 12, 36, 60, np.inf],
            labels=['new', 'established', 'loyal', 'veteran']
        )

    # Drop columns not needed for prediction
    drop_cols = ['customer', 'customer_lifetime_value', 'effective_
    feature_df = df.drop(columns=[c for c in drop_cols if c in df.

    # Handle missing values
    for col in feature_df.columns:
        if feature_df[col].dtype in ['object', 'category']:
            feature_df[col] = feature_df[col].fillna('unknown')
        else:
            feature_df[col] = feature_df[col].fillna(feature_df[co

    # Preprocess
    X_processed = preprocessor.transform(feature_df)

    # Predict (in log scale)
    log_predictions = model.predict(X_processed)

    # Convert to dollar scale
    dollar_predictions = np.expm1(log_predictions)

    # Add predictions to original data
    result_df = input_data.copy()
    result_df['Predicted_CLV_Log'] = log_predictions
    result_df['Predicted_CLV_Dollars'] = dollar_predictions.round(

    return result_df

print("■ Inference function defined")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 4. Single Customer Prediction Let's demonstrate predicting CLV for a single customer.

EXECUTION CODE:

```
# Create sample customer
print("=" * 60)
print("SINGLE CUSTOMER PREDICTION")
print("=" * 60)

sample_customer = pd.DataFrame([{
    'state': 'California',
    'response': 'No',
    'coverage': 'Premium',
    'education': 'Master',
    'employmentstatus': 'Employed',
    'gender': 'M',
    'income': 75000,
    'location_code': 'Suburban',
    'marital_status': 'Married',
    'monthly_premium_auto': 200,
    'months_since_last_claim': 12,
    'months_since_policy_inception': 48,
    'number_of_open_complaints': 0,
    'number_of_policies': 3,
    'policy_type': 'Corporate Auto',
    'renew_offer_type': 'Offer1',
    'sales_channel': 'Agent',
    'total_claim_amount': 500,
    'vehicle_class': 'Two-Door Car',
    'vehicle_size': 'Medsize'
}])

print("\n■ Sample Customer Profile:")
for col, val in sample_customer.iloc[0].items():
    print(f"   {col}: {val}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Make prediction
result = predict_clv(sample_customer, model, preprocessor)

print(f"\n■ PREDICTION RESULT:")
print(f"   Predicted CLV (Log Scale): {result['Predicted_CLV_Log']:.4f}")
print(f"   Predicted CLV (Dollars):   ${result['Predicted_CLV_Dollars'].iloc[0]:,.2f}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 5. Batch Prediction For production use, we often need to score many customers at once.

EXECUTION CODE:

```
# Load original dataset for batch prediction demo
print("=" * 60)
print("BATCH PREDICTION")
print("=" * 60)

# Load raw data
raw_data = pd.read_csv(os.path.join(DATA_RAW_DIR, 'WA_Fn-UseC_-Marketing-Customer-Value-Analysis.csv'))

# Take a sample for demonstration
sample_batch = raw_data.sample(100, random_state=42)

print(f"\n■ Scoring {len(sample_batch)} customers...")

# Make predictions
batch_results = predict_clv(sample_batch, model, preprocessor)

print(f"\n■ Batch prediction complete!")
print(f"\n■ Predicted CLV Statistics:")
print(f"   Mean:   ${batch_results['Predicted_CLV_Dollars'].mean():,.2f}")
print(f"   Median: ${batch_results['Predicted_CLV_Dollars'].median():,.2f}")
print(f"   Min:    ${batch_results['Predicted_CLV_Dollars'].min():,.2f}")
print(f"   Max:    ${batch_results['Predicted_CLV_Dollars'].max():,.2f}")
```

OBJECTIVE: We initiate the pipeline by ingesting the raw CSV. This action loads the data into memory, allowing us to perform the initial schema validation. Assessing the magnitude of the dataframe is critical at this stage.

EXECUTION CODE:

```
# Show sample results
print("\n■ Sample Predictions:")
display_cols = ['Customer', 'Customer Lifetime Value', 'Predicted_
display_cols = [c for c in display_cols if c in batch_results.colu
batch_results[display_cols].head(10)
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 6. Business Use Cases 6.1 Customer Segmentation by Predicted CLV

EXECUTION CODE:

```
# Segment customers by predicted CLV
print("=" * 60)
print("CUSTOMER SEGMENTATION BY CLV")
print("=" * 60)

# Define CLV segments
def assign_segment(clv):
    if clv >= 10000:
        return 'VIP'
    elif clv >= 6000:
        return 'High Value'
    elif clv >= 3000:
        return 'Medium Value'
    else:
        return 'Low Value'

batch_results['CLV_Segment'] = batch_results['Predicted_CLV_Dollar

# Segment distribution
segment_counts = batch_results['CLV_Segment'].value_counts()

print("\n■ Customer Segment Distribution:")
for segment, count in segment_counts.items():
    pct = count / len(batch_results) * 100
    print(f"   {segment:12} {count:4} customers ({pct:5.1f}%)")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

6.2 High-Value Customer Identification

EXECUTION CODE:

```
# Top 10 highest predicted CLV
print("=" * 60)
print("TOP 10 HIGH-VALUE CUSTOMERS")
print("=" * 60)

top_customers = batch_results.nlargest(10, 'Predicted_CLV_Dollars'
print("\n■ Highest Predicted CLV Customers:")
for i, (_, row) in enumerate(top_customers.iterrows(), 1):
    customer_id = row.get('Customer', 'N/A')
    clv = row['Predicted_CLV_Dollars']
    print(f"   {i:2}. Customer {customer_id}: ${clv:,.2f}")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

--- 7. Summary This notebook demonstrated: 1. **Loading trained artifacts** for production use 2. **Creating a reusable inference function** that handles preprocessing 3. **Single customer prediction** for real-time scoring 4. **Batch prediction** for bulk processing 5. **Business applications** including segmentation and high-value identification Deployment Considerations For production deployment, consider: - Wrapping the inference function in an API (e.g., Flask, FastAPI) - Implementing input validation and error handling - Setting up model monitoring and drift detection - Establishing a model retraining pipeline --- **End of Notebook 05 - Project Complete**

Retention Analysis Below are the high-resolution figures generated by our pipeline.

05 Retention Sweet Spot ![05 Retention Sweet Spot](../report/figures/05_retention_sweet_spot.png)

Chapter 6: The Tribes

Step 06: Customer Segmentation & Clustering "The Four Tribes of Risk" **Objective**: Beyond predicting *how much* a customer is worth (CLV), we must understand *who* they are. In this analysis, we use **K-Means Clustering** to segment our portfolio into distinct behavioral groups. **Methodology**: 1. **Feature Selection**: We selected 'Income', 'Monthly Premium', 'Total Claims', and 'Months Since Policy Inception'. 2. **Scaling**: RobustScaler was used to handle outliers. 3. **Dimensionality Reduction**: PCA was used for visualization. 4. **Algorithm**: K-Means with K=4 (determined by the Elbow Method).

EXECUTION CODE:

```
import os
from IPython.display import Image, display

# Path to figures
FIG_DIR = "../report/figures"

def show_figure(fname):
    path = os.path.join(FIG_DIR, fname)
    if os.path.exists(path.replace('../', '')): # Check existence relative to script or simple check
        display(Image(filename=path, width=800))
    else:
        # Fallback for notebook relative path
        display(Image(filename=path, width=800))
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

1. Determining the Optimal Number of Clusters We utilized the **Elbow Method** and **Silhouette Score** to identify that 4 is the optimal number of segments.

EXECUTION CODE:

```
show_figure("06_cluster_optimal_k.png")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

2. Cluster Profiles (The DNA of the Segments) The Radar Chart below reveals the distinct characteristics of each cluster. - **Cluster 0 (The Average Joe)**: Moderate income, moderate claims. - **Cluster 1 (The Power Users)**: High premium, high claims. - **Cluster 2 (The Low Value)**: Low income, low premium. - **Cluster 3 (The Profit Centers)**: High income, low claims.

EXECUTION CODE:

```
show_figure("06_cluster_radar.png")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

3. PCA Visualization Projecting the high-dimensional data into 2D space reveals the separation between the groups.

EXECUTION CODE:

```
show_figure("06_cluster_visualization.png")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

4. Deep Dive: Segment Personas Detailed histograms for each segment.

EXECUTION CODE:

```
# Segment 0
show_figure("06_cluster_seg_0.png")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Segment 1
show_figure("06_cluster_seg_1.png")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Segment 2
show_figure("06_cluster_seg_2.png")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

EXECUTION CODE:

```
# Segment 3
show_figure("06_cluster_seg_3.png")
```

OPERATION: This step transforms the internal state of the data pipeline. We execute this logical block to prepare the vectors for the subsequent analysis phase, ensuring consistency and type safety.

Conclusion & Strategy

This concludes the technical memoir. We have traversed from the raw CSV to a clustered, predictive strategic engine. The code presented herein is the exact logic used to generate the insights.