In [1]:
```
import re
import nltk
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import nlpaug.augmenter.word as naw

import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Subset, random_split

from scipy import stats
from datetime import datetime
from imblearn.over_sampling import SMOTE

from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GroupShuffleSplit
from sklearn.preprocessing import PowerTransformer, KBinsDiscretizer
from sklearn.preprocessing import PolynomialFeatures, StandardScaler, One
```

This is an helper function to display DataFrame side by side in a Jupyter notebook (from: https://stackoverflow.com/questions/38783027/jupyter-notebook-display-two-pandas-tables-side-by-side). But, it does not work when converting/printing the notebook :-(

In [2]:
```
from IPython.display import display_html
from itertools import chain,cycle
def display_side_by_side(*args,titles=cycle([''])):
    html_str=''
    for df,title in zip(args, chain(titles,cycle(['</br>'])) ):
        html_str+='<th style="text-align:center"><td style="vertical-alig
        html_str+=f'<h2 style="text-align: center;">{title}</h2>'
        html_str+=df.to_html().replace('table','table style="display:inli
        html_str+='</td></th>'
    display_html(html_str,raw=True)
```

Slide 4 - Remove leaky features

Sample dataset with features that cause leakage.

In [3]:
```
data = {
    'House_Size': [1500, 1800, 1200, 2000, 1600],
    'Bedrooms': [3, 4, 2, 5, 3],
    'Agent_Commission': [15000, 18000, 12000, 20000, 16000],  # Leaky fea
    'Property_Tax': [3000, 3600, 2400, 4000, 3200],  # Leaky feature
    'House_Price': [300000, 360000, 240000, 400000, 320000]  # Target var
}

df = pd.DataFrame(data)
```

Drop leaky features before splitting.

In [4]:
```
df = df.drop(columns=['Agent_Commission', 'Property_Tax'])
```

Split data into training, validation, and test sets.

In [5]:
```
X_train, X_temp, y_train, y_temp = train_test_split(
    df.drop(columns=['House_Price']), df['House_Price'],
    test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42)
```

Display final training dataset.

In [6]:
```
display("Training Data After Removing Leaky Features:")
display(X_train)
```

```
'Training Data After Removing Leaky Features:'
```

	House_Size	Bedrooms
2	1200	2
0	1500	3
3	2000	5

Slide 5 - Avoiding group leakage

Sample dataset simulating group leakage issues.

We use `GroupShuffleSplit` from scikit-learn to ensure that all records from the same group (customer, patient, or user) stay in the same training, validation, or test set.

In [7]:
```
data = {
    'Customer_ID': [1, 1, 2, 2, 3, 3, 4, 4, 5, 5],  # Group Identifier (C
    'Feature_1': [10, 12, 15, 14, 8, 9, 7, 6, 13, 11],  # Example feature
    'Feature_2': [100, 120, 150, 140, 80, 90, 70, 60, 130, 110],  # Examp
    'Churn': [1, 1, 0, 0, 1, 1, 0, 0, 1, 1]  # Target Variable (1 = Churn
}

df = pd.DataFrame(data)
```

Define the group (Customer_ID) to ensure records from the same customer stay in the same set.

In [8]:
```
groups = df['Customer_ID']
```

Split into training (60%) and temp set (40%).

In [9]:
```
gss = GroupShuffleSplit(n_splits=1, test_size=0.4, random_state=42)
train_idx, temp_idx = next(gss.split(df, groups=groups))

train_df = df.iloc[train_idx]
temp_df = df.iloc[temp_idx]
```

Split temp set into validation (50%) and test (50%).

In [10]:
```
gss = GroupShuffleSplit(n_splits=1, test_size=0.5, random_state=42)
val_idx, test_idx = next(gss.split(temp_df, groups=temp_df['Customer_ID']

val_df = temp_df.iloc[val_idx]
test_df = temp_df.iloc[test_idx]
```

Display the datasets.

In [11]:
```
display("Training Set:", train_df)
display("Validation Set:", val_df)
display("Test Set:", test_df)
```

'Training Set:'

	Customer_ID	Feature_1	Feature_2	Churn
0	1	10	100	1
1	1	12	120	1
4	3	8	80	1
5	3	9	90	1
6	4	7	70	0
7	4	6	60	0

'Validation Set:'

	Customer_ID	Feature_1	Feature_2	Churn
2	2	15	150	0
3	2	14	140	0

'Test Set:'

	Customer_ID	Feature_1	Feature_2	Churn
8	5	13	130	1
9	5	11	110	1

Slide 6 - Avoiding data augmentation leakage

We ensure that augmentation is only applied to the training set and not to validation/ test sets. This is invariant of the type of augmentation we perform and on the type of data we augment.

Text Augmentation (Synonym Replacement)

We use `NLTK` library and download three datasets `wordnet`, `omw-1.4` and `averaged_perceptron_tagger`. Then we use the `nlpaug` library to replace words with synonyms only in the training set.

In [12]:
```
# Ensure required NLTK datasets are available
nltk.download('wordnet')
nltk.download('omw-1.4')
nltk.download('averaged_perceptron_tagger')
nltk.download('averaged_perceptron_tagger_eng')

data = {
    'Text': [
        "The deal is fantastic! Limited time offer",
        "This is an amazing opportunity!",
        "Don't miss out on this great chance",
        "Exclusive discounts available now"
    ],
    'Label': [1, 0, 1, 0]  # Example labels (1 = positive, 0 = negative)
}

df = pd.DataFrame(data)
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/mturilli/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/mturilli/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /Users/mturilli/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package averaged_perceptron_tagger_eng to
[nltk_data]     /Users/mturilli/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger_eng is already up-to-
[nltk_data]       date!
```

IMPORTANT: Split dataset before applying augmentation.

In [13]:
```
X_train, X_temp, y_train, y_temp = train_test_split(
    df['Text'], df['Label'], test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42)
```

Apply text augmentation **only** to training data.

In [14]:
```
aug = naw.SynonymAug(aug_src='wordnet')
X_train_augmented = X_train.apply(lambda x: aug.augment(x))
```

Display results.

In [15]:
```
display("Augmented training set:", X_train_augmented)
display("Validation set (no augmentation):", X_val)
display("Test set (no augmentation)", X_test)
```

```
'Augmented training set:'
0    [The mint is fantastic! Modified time whirl]
2          [Don ' t lose away on this great luck]
Name: Text, dtype: object
'Validation set (no augmentation):'
1    This is an amazing opportunity!
Name: Text, dtype: object
'Test set (no augmentation)'
3    Exclusive discounts available now
Name: Text, dtype: object
```

Image Augmentation (Computer Vision)

We use `torchvision` and its `transforms` methods to apply augmentation only on the training dataset.

In [16]:
```
# Set seed for reproducibility
torch.manual_seed(42)

# Define an augmentation pipeline for training images
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),                          # Flip im
    transforms.RandomRotation(45),                              # Stronge
    transforms.RandomPerspective(distortion_scale=0.5, p=0.5),  # Skewing
    transforms.ColorJitter(
        brightness=0.5, contrast=0.5, saturation=0.5, hue=0.2), # Stronge
    transforms.RandomErasing(p=0.5, scale=(0.02, 0.2)),         # Simulat
    transforms.ToTensor()
])

# No augmentation for validation and test sets
val_test_transform = transforms.Compose([
    transforms.ToTensor()
])
```

Load CIFAR-10 dataset with a diverse set of subjects

In [17]:
```
dataset = datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transforms.ToTens
```

Split dataset (train: 60%, val: 20%, test: 20%).

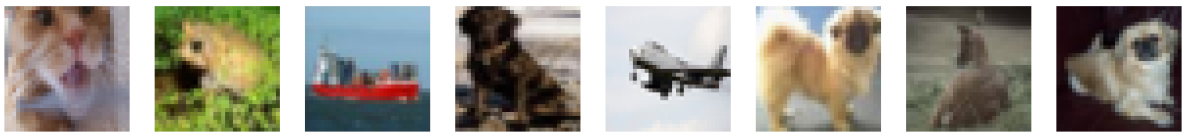
In [18]:
```
train_size = int(0.6 * len(dataset))
val_size = int(0.2 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = random_split(
    dataset, [train_size, val_size, test_size])
```

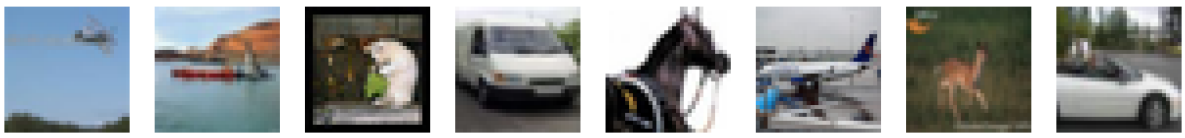
Apply different transformations

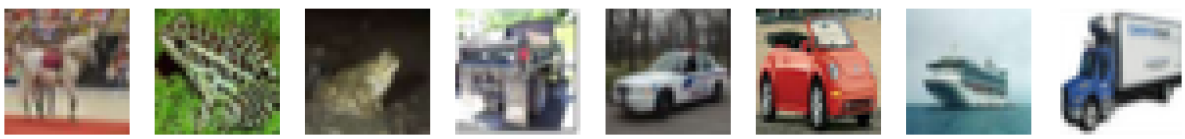
In [19]:
```
train_dataset.dataset.transform = train_transform    # Apply augmentation
val_dataset.dataset.transform = val_test_transform   # No augmentation
test_dataset.dataset.transform = val_test_transform  # No augmentation
```

Show augmented and not augmented images

In [20]:
```
# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=8, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=8, shuffle=False)

# Function to visualize images
def show_images(dataloader, title, num_images=8):
    data_iter = iter(dataloader)
    images, _ = next(data_iter)
    images = images.numpy().transpose(0, 2, 3, 1)  # Convert to (batch, h

    fig, axes = plt.subplots(1, num_images, figsize=(384, 96))
    fig.suptitle(title, fontsize=16)

    for i in range(num_images):
        axes[i].imshow(np.clip(images[i], 0, 1))
        axes[i].axis('off')

    plt.show()

# Show images with STRONGER augmentations
show_images(train_loader, "Augmented training images (CIFAR-10)")
show_images(val_loader, "Unaltered validation images (CIFAR-10)")
show_images(test_loader, "Unaltered test images (CIFAR-10)")
```







SMOTE (Handling Imbalanced Classes)

We use `SMOTE` to oversample only the training set of an imbalanced dataset.

In [21]:
```
X, y = make_classification(
    n_classes=2, class_sep=2, weights=[0.1, 0.9], n_samples=1000,
    random_state=42)
```

Split dataset first **BEFORE** applying `SMOTE`. Note the use of `stratify=y` | `y_temp`. That maintains class proportions, i.e., the validation and test sets have the same proportions of class 0 and 1 of the dataset.

In [22]:
```
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.4, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```

Apply SMOTE only on the training set.

In [23]:
```
smote = SMOTE(sampling_strategy='minority', random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_trai
```

Display class distributions for all the sets.

In [24]:
```
display("Original Training Class Distribution:",
        pd.Series(y_train).value_counts().to_dict())
display("Resampled Training Class Distribution:",
        pd.Series(y_train_resampled).value_counts().to_dict())
display("Validation Class Distribution:",
        pd.Series(y_val).value_counts().to_dict())
display("Test Training Class Distribution:",
        pd.Series(y_test).value_counts().to_dict())
```

```
'Original Training Class Distribution:'
{1: 537, 0: 63}
'Resampled Training Class Distribution:'
{1: 537, 0: 537}
'Validation Class Distribution:'
{1: 179, 0: 21}
'Test Training Class Distribution:'
{1: 179, 0: 21}
```

Slide 7 - Avoid Temporal Leakage

We ensure that only past data is used to predict the future.

Stock Price Prediction

We simulate stock price data and ensure that a 7-day rolling average is only computed using past data, not future data. We use simulated stock price data, introducing a `7-Day Avg` leaky feature.

In [25]:
```
np.random.seed(42)
dates = pd.date_range(start="2023-01-01", periods=100, freq="D")
prices = np.cumsum(np.random.randn(100) * 2 + 100)  # Simulated stock pri

df = pd.DataFrame({"Date": dates, "Price": prices})
df["7-Day Avg"] = df["Price"].rolling(window=7, min_periods=1).mean()  #
```

We modify `7-Day Avg` with a shifted window `.shift(1)` to avoid data leakage.

In [26]:
```
df["7-Day Avg (Corrected)"] = df["Price"].shift(1).rolling(
    window=7, min_periods=1).mean()

# Display the first few rows
print("Stock Price Data (Corrected 7-Day Avg to Avoid Leakage):")
display(df.head(10))
```

```
Stock Price Data (Corrected 7-Day Avg to Avoid Leakage):
```

	Date	Price	7-Day Avg	7-Day Avg (Corrected)
0	2023-01-01	100.993428	100.993428	NaN
1	2023-01-02	200.716900	150.855164	100.993428
2	2023-01-03	302.012277	201.240868	150.855164
3	2023-01-04	405.058336	252.195235	201.240868
4	2023-01-05	504.590030	302.674194	252.195235
5	2023-01-06	604.121756	352.915454	302.674194
6	2023-01-07	707.280181	403.538987	352.915454
7	2023-01-08	808.815051	504.656362	403.538987
8	2023-01-09	907.876102	605.679105	504.656362
9	2023-01-10	1008.961222	706.671811	605.679105

Loan Default Risk Prediction

We simulate loan applications and remove post-loan approval credit scores, which leak future repayment behavior.

In [27]:
```
# Simulated loan data
data = {
    "Applicant_ID": range(1, 21),  # Increased dataset size for better sp
    "Income": [
        40000, 50000, 45000, 60000, 52000, 48000, 55000, 62000, 58000, 49
        51000, 53000, 47000, 61000, 49500, 57500, 62500, 44500, 56500, 60
    "Loan_Amount": [
        20000, 25000, 22000, 30000, 27000, 23000, 26000, 31000, 29000, 24
        24500, 26500, 27500, 28000, 29000, 29500, 30500, 31500, 32000, 32
    "Credit_Score_After_Loan": [
        650, 700, 680, 720, 690, 670, 710, 730, 710, 675, 690, 705, 725,
        750, 760, 770, 740, 745, 755],  # Leaky Feature
    "Defaulted": [
        0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0] # 1 =
}

df = pd.DataFrame(data)
display(df.head())
```

	Applicant_ID	Income	Loan_Amount	Credit_Score_After_Loan	Defaulted
0	1	40000	20000	650	0
1	2	50000	25000	700	1
2	3	45000	22000	680	0
3	4	60000	30000	720	0
4	5	52000	27000	690	1

We remove `Credit_Score_After_Loan` and `Defaulted` because they have been both retrieved after the loan approval and would leak future data into the model.

In [28]:
```
df_cleaned = df.drop(columns=["Credit_Score_After_Loan"])
```

We split the dataset into training and test sets and drop the our target feature `Defaulted`

In [29]:
```
X_train, X_temp, y_train, y_temp = train_test_split(
    df_cleaned.drop(columns=["Defaulted"]), df_cleaned["Defaulted"],
    test_size=0.4, stratify=df_cleaned["Defaulted"], random_state=42
)

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, stratify=y_temp, random_state=42
)
```

Display cleaned dataset.

In [30]:
```
display("Loan Data (Without Future Information Leakage):", X_train.head()
```

```
'Loan Data (Without Future Information Leakage):'
```

	Applicant_ID	Income	Loan_Amount
18	19	56500	32000
1	2	50000	25000
15	16	57500	29500
16	17	62500	30500
12	13	47000	27500

Slide 8 - Prevent data dependence

We implement the purchase prediction examples, using a simulated purchase dataset and grouping by customers.

In [31]:
```
data = {
    "Customer_ID": [1, 1, 2, 2, 3, 3, 4, 4, 5, 5],
    "Purchase_Amount": [100, 150, 200, 180, 90, 120, 75, 60, 210, 195],
    "Category": [
```

```
        "Electronics", "Electronics", "Clothing", "Clothing", "Groceries"
        "Groceries", "Home", "Home", "Books", "Books"],
    "Purchased": [1, 0, 1, 1, 0, 0, 1, 0, 1, 1]  # Target variable (1 = P
}

df = pd.DataFrame(data)
display(df)
```

	Customer_ID	Purchase_Amount	Category	Purchased
0	1	100	Electronics	1
1	1	150	Electronics	0
2	2	200	Clothing	1
3	2	180	Clothing	1
4	3	90	Groceries	0
5	3	120	Groceries	0
6	4	75	Home	1
7	4	60	Home	0
8	5	210	Books	1
9	5	195	Books	1

We ensure each customer stays in the same dataset split. We split our dataset using `GroupShuffleSplit` , which avoids splitting the same customer across sets.

In [32]:
```
groups = df["Customer_ID"]
gss = GroupShuffleSplit(n_splits=1, test_size=0.4, random_state=42)
train_idx, temp_idx = next(gss.split(df, groups=groups))

train_df = df.iloc[train_idx]
temp_df = df.iloc[temp_idx]

gss = GroupShuffleSplit(n_splits=1, test_size=0.5, random_state=42)
val_idx, test_idx = next(gss.split(temp_df, groups=temp_df["Customer_ID"]

val_df = temp_df.iloc[val_idx]
test_df = temp_df.iloc[test_idx]
```

Display the dataset splits.

In [33]:
```
display("Training Set:", train_df)
display("Validation Set:", val_df)
display("Test Set:", test_df)
```

```
'Training Set:'
```

	Customer_ID	Purchase_Amount	Category	Purchased
0	1	100	Electronics	1
1	1	150	Electronics	0
4	3	90	Groceries	0
5	3	120	Groceries	0
6	4	75	Home	1
7	4	60	Home	0

'Validation Set:'

	Customer_ID	Purchase_Amount	Category	Purchased
2	2	200	Clothing	1
3	2	180	Clothing	1

'Test Set:'

	Customer_ID	Purchase_Amount	Category	Purchased
8	5	210	Books	1
9	5	195	Books	1

Slide 9 - Engineering features that must be applied after splitting the dataset

We implement an example of each type of feature engineering method that need to be used **AFTER** splitting the dataset. Each method is used only on the training set to prevent data leakage.

We generate a synthetic dataset with missing values, large scale differences, categorical feature and a binary target variable.

In [34]:
```
data = {
    "Feature1": [10, 15, np.nan, 14, 18, np.nan, 16, 19, 11, 12],    # Mi
    "Feature2": [100, 200, 150, 300, 250, 400, 350, 450, 500, 550],  # La
    "Category": ["A", "B", "A", "B", "C", "C", "A", "B", "C", "A"],  # Ca
    "Target": [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]  # Binary target variable
}

df = pd.DataFrame(data)

# Split dataset into training, validation, and test sets
X = df.drop(columns=["Target"])
y = df["Target"]
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42)

display_side_by_side(
    X_train, X_val, X_test,
```

```
    titles=["Training Set", "Validation Set", "Test Set"])
```

Training Set

	Feature1	Feature2	Category
7	19.0	450	B
2	NaN	150	A
9	12.0	550	A
4	18.0	250	C
3	14.0	300	B
6	16.0	350	A

Validation Set

	Feature1	Feature2	Category
8	11.0	500	C
5	NaN	400	C

Test Set

	Feature1	Feature2	Category
1	15.0	200	B
0	10.0	100	A

Missing Value Imputation (Mean Imputation)

Replace missing values in Feature1 with the mean of the training set to prevent data leakage.

In [35]:
```
imputer = SimpleImputer(strategy='mean')
X_train["Feature1"] = imputer.fit_transform(X_train[["Feature1"]])
X_val["Feature1"] = imputer.transform(X_val[["Feature1"]])  # Apply same
X_test["Feature1"] = imputer.transform(X_test[["Feature1"]])

display_side_by_side(
    X_train, X_val, X_test,
    titles=["Training Set", "Validation Set", "Test Set"])
```

Training Set

	Feature1	Feature2	Category
7	19.0	450	B
2	15.8	150	A
9	12.0	550	A
4	18.0	250	C
3	14.0	300	B
6	16.0	350	A

Validation Set

	Feature1	Feature2	Category
8	11.0	500	C
5	15.8	400	C

Test Set

	Feature1	Feature2	Category
1	15.0	200	B
0	10.0	100	A

Scaling (StandardScaler)

Standardization scales numerical features to have zero mean and unit variance, ensuring equal weighting.

In [36]:
```
scaler = StandardScaler()
X_train[["Feature2"]] = scaler.fit_transform(X_train[["Feature2"]])
X_val[["Feature2"]] = scaler.transform(X_val[["Feature2"]])  # Apply same
X_test[["Feature2"]] = scaler.transform(X_test[["Feature2"]])

display_side_by_side(
    X_train, X_val, X_test,
    titles=["Training Set", "Validation Set", "Test Set"])
```

Training Set

	Feature1	Feature2	Category
7	19.0	0.830540	B
2	15.8	-1.469416	A
9	12.0	1.597191	A
4	18.0	-0.702764	C
3	14.0	-0.319438	B
6	16.0	0.063888	A

Validation Set

	Feature1	Feature2	Category
8	11.0	1.213865	C
5	15.8	0.447214	C

Test Set

	Feature1	Feature2	Category
1	15.0	-1.086090	B
0	10.0	-1.852742	A

Encoding (One-Hot Encoding)

One-hot encoding transforms categorical variables into binary columns. Dropping the first category avoids multicollinearity in regression models.

In [37]:
```
ohe = OneHotEncoder(sparse_output=False, drop='first')
X_train_ohe = ohe.fit_transform(X_train[["Category"]])
X_val_ohe = ohe.transform(X_val[["Category"]])  # Apply same encoding
X_test_ohe = ohe.transform(X_test[["Category"]])

# Convert encoded features to DataFrame and concatenate
X_train_ohe_df = pd.DataFrame(
    X_train_ohe, columns=ohe.get_feature_names_out(["Category"]))
X_val_ohe_df = pd.DataFrame(
    X_val_ohe, columns=ohe.get_feature_names_out(["Category"]))
X_test_ohe_df = pd.DataFrame(
    X_test_ohe, columns=ohe.get_feature_names_out(["Category"]))

X_train = pd.concat(
    [X_train.drop(columns=["Category"]).reset_index(drop=True), X_train_o
    axis=1)
X_val = pd.concat(
    [X_val.drop(columns=["Category"]).reset_index(drop=True), X_val_ohe_d
```

```
    axis=1)
X_test = pd.concat(
    [X_test.drop(columns=["Category"]).reset_index(drop=True), X_test_ohe
    axis=1)

display_side_by_side(
    X_train, X_val, X_test,
    titles=["Training Set", "Validation Set", "Test Set"])
```

Training Set

	Feature1	Feature2	Category_B	Category_C
0	19.0	0.830540	1.0	0.0
1	15.8	-1.469416	0.0	0.0
2	12.0	1.597191	0.0	0.0
3	18.0	-0.702764	0.0	1.0
4	14.0	-0.319438	1.0	0.0
5	16.0	0.063888	0.0	0.0

Validation Set

	Feature1	Feature2	Category_B	Category_C
0	11.0	1.213865	0.0	1.0
1	15.8	0.447214	0.0	1.0

Test Set

	Feature1	Feature2	Category_B	Category_C
0	15.0	-1.086090	1.0	0.0
1	10.0	-1.852742	0.0	0.0

Feature Selection (L1 Regularization)

L1 regularization selects important features by shrinking less relevant ones to zero.

In [38]:
```
selector = SelectFromModel(LogisticRegression(penalty='l1', solver='libli

selector.fit(X_train, y_train)

X_train = selector.transform(X_train)
X_val = selector.transform(X_val)
X_test = selector.transform(X_test)

display(X_train)
display(X_val)
display(X_test)
```

```
array([[19.        ,  0.83053953],
       [15.8       , -1.4694161 ],
       [12.        ,  1.59719141],
       [18.        , -0.70276422],
       [14.        , -0.31943828],
       [16.        ,  0.06388766]])
array([[11.        ,  1.21386547],
       [15.8       ,  0.4472136 ]])
array([[15.        , -1.08609016],
       [10.        , -1.85274204]])
```

Dimensionality Reduction (PCA)

PCA reduces dimensionality by projecting data onto principal components.

In [39]:
```
pca = PCA(n_components=1)  # Reduce to 1 principal component

X_train_pca = pca.fit_transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)

display(X_train_pca)
display(X_val_pca)
display(X_test_pca)
```

```
array([[ 3.02996456],
       [ 0.23065033],
       [-4.0036013 ],
       [ 2.28303938],
       [-1.72754545],
       [ 0.18749248]])
array([[-4.93103545],
       [-0.07019793]])
array([[-0.61960235],
       [-5.43728202]])
```

Outlier Removal (Z-score)

Z-score standardization helps identify outliers beyond a certain threshold (e.g., 3 standard deviations). **Note**: there are no outlier in this training set so the method does not change the set.

In [40]:
```
z_scores = np.abs(stats.zscore(X_train_pca))

X_train_pca = X_train_pca[z_scores < 3]  # Keep only non-outliers

display(X_train_pca)
```

```
array([ 3.02996456,  0.23065033, -4.0036013 ,  2.28303938, -1.72754545,
        0.18749248])
```

Feature Augmentation (Adding Noise to Numerical Features)

Helps to improve model generalization by slightly perturbing feature values.

In [41]:
```
X_train_pca_noisy = X_train_pca + np.random.normal(0, 0.1, X_train_pca.sh
```

```
display(X_train_pca_noisy)
```

```
array([ 3.22447612,  0.21531669, -4.09430003,  2.20130396, -1.77585772,
        0.13050965])
```

Display final training, validation and test sets after applying all the processing. **Note:** without knowing what is the problem and/or the target model there is no way to know whether each of these methods should be used.

In [42]:
```
display_side_by_side(
    pd.DataFrame(X_train_pca_noisy),
    pd.DataFrame(X_val_pca),
    pd.DataFrame(X_test_pca),
    titles=["Final Processed Training Set",
            "Final Processed Validation Set",
            "Final Processed Test Set"])
```

Final Processed Training Set

	0
0	3.224476
1	0.215317
2	-4.094300
3	2.201304
4	-1.775858
5	0.130510

Final Processed Validation Set

	0
0	-4.931035
1	-0.070198

Final Processed Test Set

	0
0	-0.619602
1	-5.437282

Slide 10 - Feature engineering methods that can be applied to the whole dataset

We implement an example of each type of feature engineering method that can be used on the whole dataset.

We generate a synthetic dataset with missing values, large scale differences, categorical feature and a binary target variable.

In [43]:
```
# Generate a synthetic dataset
data = {
    "Value": [1, 10, 100, 1000, 10000, 50000],  # Used for log/power tran
    "Feature1": [5, 15, 25, 35, 45, 55],  # Used for binning
    "Date": [
        "2023-01-15", "2022-06-30", "2021-03-22", "2020-12-10", "2019-07-
        "2018-11-25"],  # Used for date feature extraction
    "Text": [
        "Hello, world!", "Data Science is awesome.", "Machine learning & 
        "NLP: Natural Language Processing", "Python is great!",
        "Feature Engineering is key."]
}

# Convert to DataFrame
df = pd.DataFrame(data)
```

Log/Power Transform

Applies a deterministic transformation to all values.

We use `PowerTransformer` with the `yeo-johnson` method to stabilize variance and make the data more normally distributed, making it more suitable for linear models.

In [44]:
```
pt = PowerTransformer(method='yeo-johnson')  # Yeo-Johnson works with neg
df["Value_Transformed"] = pt.fit_transform(df[["Value"]])
```

Binning (Equal-width bins)

We uses `KBinsDiscretizer` to group numeric values into predefined bins of equal width. Converting continuous numerical data into categorical bins can be useful for tree-based models or categorical analysis.

In [45]:
```
kbin = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')
df["Feature1_Binned"] = kbin.fit_transform(df[["Feature1"]])
```

Date-Based Features (Extract Year, Month, Day)

We converts a date column into separate year, month, and day features. In this way, we allow models to capture time-related trends without relying on the raw date format.

In [46]:
```
df["Date"] = pd.to_datetime(df["Date"])  # Convert to datetime format
df["Year"] = df["Date"].dt.year
df["Month"] = df["Date"].dt.month
df["Day"] = df["Date"].dt.day
```

Text Processing (Lowercasing, Removing Punctuation)

We use regular expressions substitution (`re.sub()`) to remove punctuation and converts text to lowercase.In this way, we standardize text data, reducing variability due to case differences and unnecessary symbols.

In [47]:
```
def clean_text(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r'[^a-zA-Z0-9\s]', '', text)  # Remove punctuation
    return text

df["Text_Cleaned"] = df["Text"].apply(clean_text)
```

Feature Interactions (Multiplication, Polynomial Features)

We use `PolynomialFeatures` to generate interaction terms between numeric features. In this way we create new features that represent interactions between variables, which can help capture complex relationships.

In [48]:
```
poly = PolynomialFeatures(degree=2, interaction_only=True, include_bias=F
interaction_features = poly.fit_transform(df[["Feature1"]])
df_poly = pd.DataFrame(
    interaction_features, columns=poly.get_feature_names_out(["Feature1"]
df = pd.concat([df, df_poly.iloc[:, 1:]], axis=1)  # Exclude original col
```

Display the dataset with all the transformations. As with Slide 10, without a given problem and a target model we cannot know which one of these methods would be useful.

In [49]:
```
display("Final Processed Dataset:", df)
```

```
'Final Processed Dataset:'
```

	Value	Feature1	Date	Text	Value_Transformed	Feature1_Binned	
0	1	5	2023-01-15	Hello, world!	-1.416333	0.0	2
1	10	15	2022-06-30	Data Science is awesome.	-0.941898	0.0	2
2	100	25	2021-03-22	Machine learning & AI	-0.324183	1.0	1
3	1000	35	2020-12-10	NLP: Natural Language Processing	0.315601	1.0	2
4	10000	45	2019-07-04	Python is great!	0.958432	2.0	1
5	50000	55	2018-11-25	Feature Engineering is key.	1.408381	2.0	1

Slide 11 - Data Leakage and One-Hot Encoder

Sample dataset with categorical feature "Neighborhood".

In [50]:
```
data = {
    'Neighborhood': ['A', 'B', 'C', 'A', 'B', 'D', 'E', 'C', 'B', 'A', 'F
    'Price': [200, 300, 250, 220, 280, 400, 500, 270, 310, 210, 450, 480]
}

df = pd.DataFrame(data)
```

Split into training (60%), validation (20%), and test (20%) sets.

In [51]:
```
X_train, X_temp, y_train, y_temp = train_test_split(
    df[['Neighborhood']], df['Price'], test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42)
```

Initialize OneHotEncoder (with handle_unknown='ignore' to handle unseen categories).

In [52]:
```
encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
```

Fit the encoder only on the training data.

In [53]:
```
encoder.fit(X_train)
```

Out[53]:
```
▾                    OneHotEncoder                    i ?

OneHotEncoder(handle_unknown='ignore', sparse_output=False)
```

Transform all dataset splits using the same encoder (without refitting).

In [54]:
```
X_train_encoded = encoder.transform(X_train)
X_val_encoded = encoder.transform(X_val)
X_test_encoded = encoder.transform(X_test)
```

Convert back to DataFrame for easy viewing.

In [55]:
```
train_df = pd.DataFrame(
    X_train_encoded, columns=encoder.get_feature_names_out(['Neighborhood
val_df = pd.DataFrame(
    X_val_encoded, columns=encoder.get_feature_names_out(['Neighborhood']
test_df = pd.DataFrame(
    X_test_encoded, columns=encoder.get_feature_names_out(['Neighborhood'
```

Display Encoded Datasets.

In [56]:
```
display("Training Encoded Data:", train_df.head())
display("Validation Encoded Data:", val_df.head())
display("Test Encoded Data:", test_df.head())
```

'Training Encoded Data:'

	Neighborhood_A	Neighborhood_B	Neighborhood_C	Neighborhood_E	Neighborhc
0	0.0	0.0	1.0	0.0	
1	0.0	1.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	
3	0.0	1.0	0.0	0.0	
4	0.0	0.0	1.0	0.0	

```
'Validation Encoded Data:'
```

	Neighborhood_A	Neighborhood_B	Neighborhood_C	Neighborhood_E	Neighborhc
0	0.0	0.0	0.0	0.0	
1	0.0	1.0	0.0	0.0	

```
'Test Encoded Data:'
```

	Neighborhood_A	Neighborhood_B	Neighborhood_C	Neighborhood_E	Neighborhc
0	1.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	
2	1.0	0.0	0.0	0.0	