Lesson 7: Data Transform

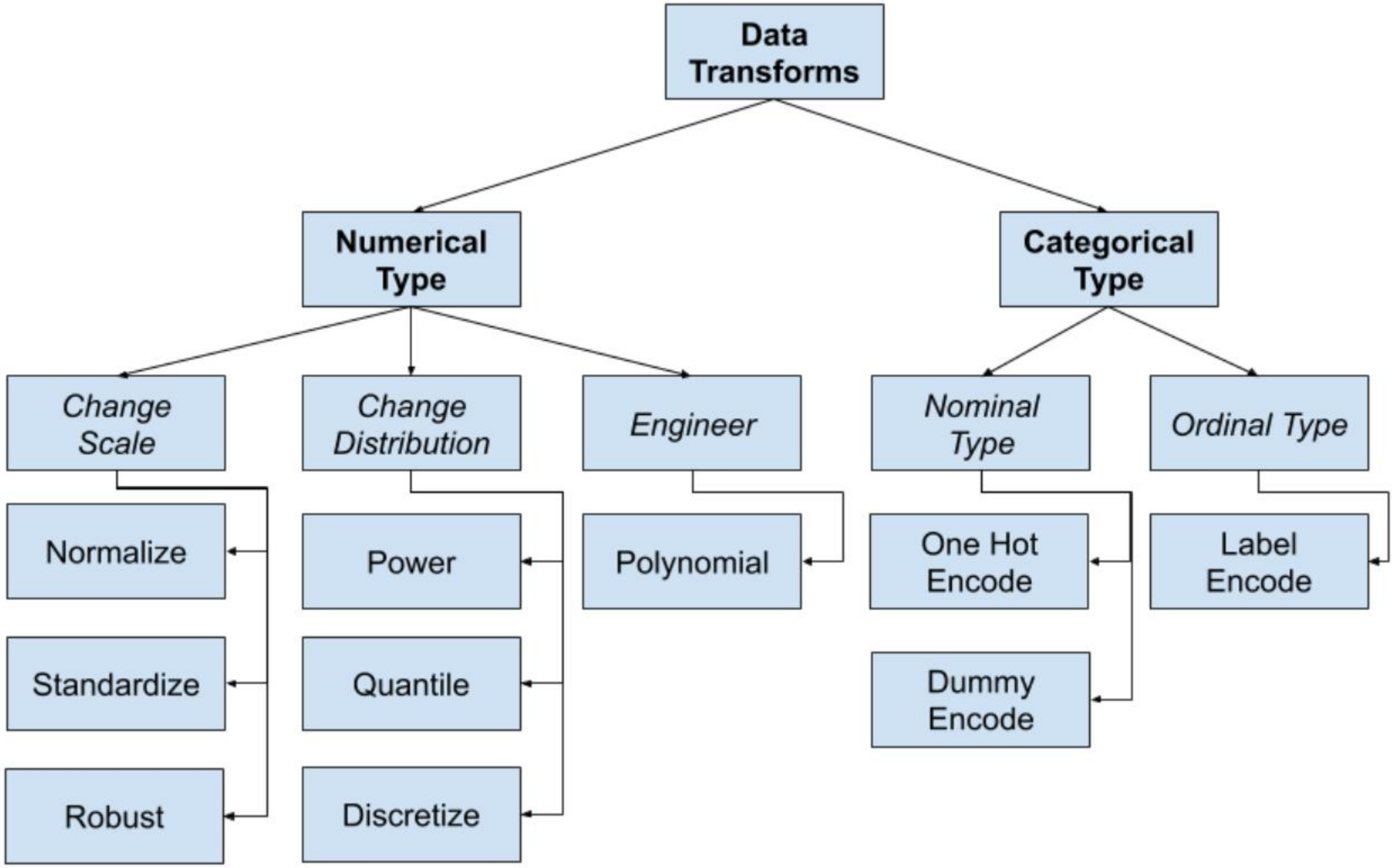
Lecturer: **Dr. Nguyen Tuan Long**
Email: ntlong@neu.edu.vn
Mobile: 0982 746 235

Why Do We Need to Transform Data?

- **Model Expectations:** Most ML algorithms don't work well with raw data.
- **Data Types:** Algorithms require numerical input, not text or categories.
- **Scale Sensitivity:** Models like SVM, KNN, and Linear Regression are sensitive to the scale of features. Large-scale features can dominate others.
  - *Example: Income (e.g., 50,000) vs. Years of Experience (e.g., 5)*
- **Distribution Assumptions:** Some models (e.g., Linear Regression) perform best when data follows a Gaussian (normal) distribution.

Goal: Convert raw data into a clean, well-structured format that improves model performance.

Overview of Data Transform



Agenda

Part I: Feature Transformation

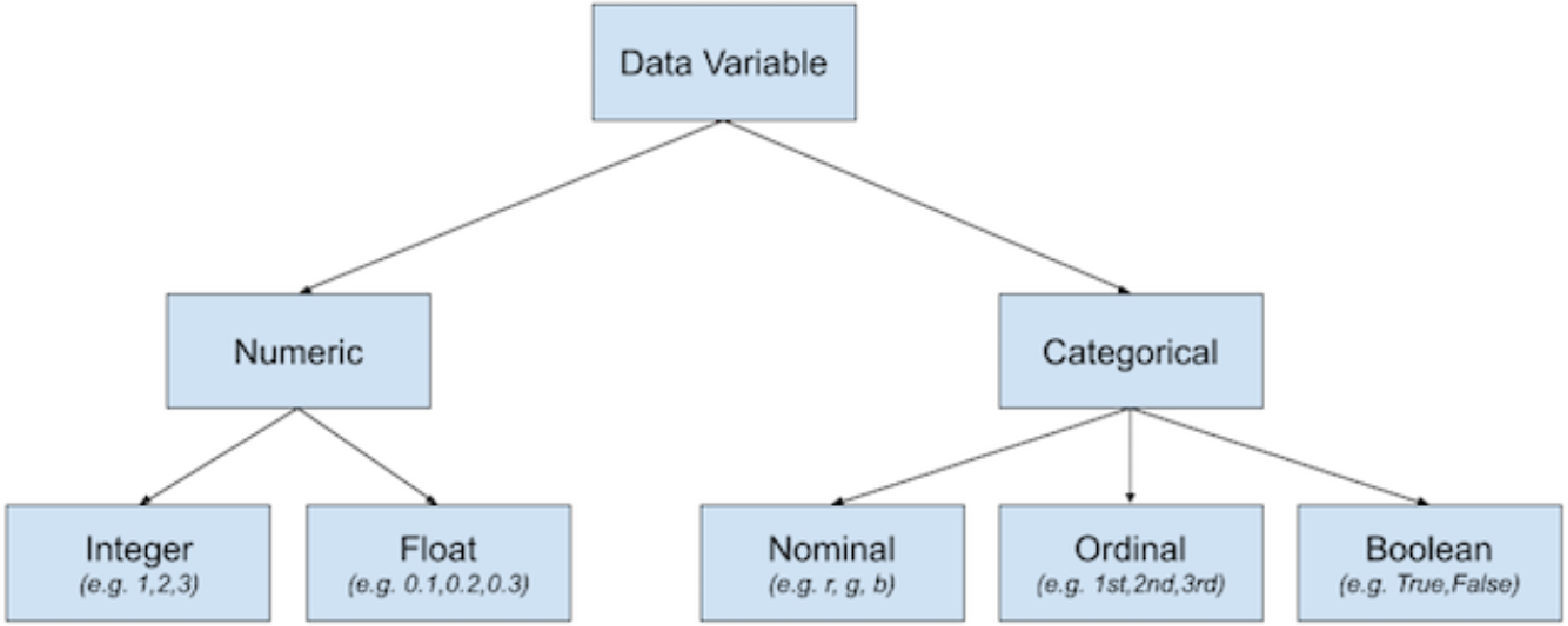
- Scaling Numerical Data
- Encoding Categorical Data

Part II: Distribution & Other Transforms

- Power & Quantile Transforms
- Discretization & Polynomial Features
- Transforming the Target Variable

Part I: Feature Transform

Overview of Data Variable Types



Part I - Scaling Numerical Data

Problem: Features are on different scales.

Solution: Bring all features to a common scale.

Key Techniques:

- MinMaxScaler (Normalization)
- StandardScaler (Standardization)
- RobustScaler

Scaling Technique 1: MinMaxScaler

Definition: MinMaxScaler (also known as Normalization) is a method of scaling data to bring all values into a specified range, typically $[0, 1]$.

- **Formula:**

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Where: - X is the original value.

- X_{min} is the minimum value of the data column.

- X_{max} is the maximum value of the data column.

- **When to Use:**
  - Algorithms without strong assumptions on data distribution (e.g., KNN).
  - Image processing (pixel values 0-255).
- **Major Drawback: Very sensitive to outliers.** An extreme value can squash all other data points into a tiny range.

Scaling Technique 1: MinMaxScaler

Usage with sklearn

```python
from sklearn.preprocessing import MinMaxScaler

# Data needs to be reshaped into a column vector (-1, 1)
data_reshaped = data.reshape(-1, 1)

scaler = MinMaxScaler()
scaled_data_sklearn = scaler.fit_transform(data_reshaped)
```

Question: What if the new data (test set) has values outside the $[\min, \max]$ range of the training data?

Scaling Technique 2: StandardScaler

Definition: StandardScaler (also known as Standardization) transforms the data so that it has a distribution with a mean of 0 and a standard deviation of 1.

Formula:

$$X_{scaled} = \frac{X - \mu}{\sigma}$$

Where: X is the original value; μ is the mean of the data column; σ is the standard deviation of the data column.

When to Use:

- The default choice for most problems.
- Algorithms that assume a Gaussian distribution (Linear Regression, Logistic Regression).

Important: It does not change the shape of the distribution, it only centers and scales it.

Scaling Technique 2: StandardScaler

Usage with sklearn

```
from sklearn.preprocessing import StandardScaler

data_reshaped = data.reshape(-1, 1)

scaler = StandardScaler()
scaled_data_sklearn =
scaler.fit_transform(data_reshaped)
```

Question: Does it make a skewed distribution become normal?

Scaling Technique 3: RobustScaler

What it does: Scales data using statistics that are robust to outliers.

- Uses the **median** and **Interquartile Range (IQR)** instead of mean and standard deviation.

Formula:

$$X_{scaled} = \frac{X - Q_2(X)}{Q_3(X) - Q_1(X)}$$

Where: $Q_2(X)$ is the median (50th percentile), $Q_1(X)$ is the 25th percentile, $Q_3(X)$ is the 75th percentile, $Q_3(X) - Q_1(X)$ is the **IQR**.

When to Use:

- **The best choice when your data has many outliers.** It prevents extreme values from skewing the scaling of other data points.

Scaling Technique 3: RobustScaler

Usage with `sklearn`

```
from sklearn.preprocessing import RobustScaler

data_reshaped = data_with_outlier.reshape(-1, 1)

scaler = RobustScaler()
scaled_data_sklearn = scaler.fit_transform(data_reshaped)
```

Question: How do the results of the 3 scalers compare on the same data with an outlier?

Scaling - Which One to Choose?

Scaler	**Best For**	**Key Characteristic**
MinMaxScaler	Data without outliers, KNN	Scales to a fixed range (e.g., [0,1]).
StandardScaler	Most general cases, linear models	Centers at mean=0, std=1.
RobustScaler	Data with outliers	Uses median and IQR to resist outliers.

Golden Rule: ALWAYS fit the scaler on the **training data only** and use it to transform both the training and test data. Never fit on the full dataset at once! This prevents **data leakage**.

Lab #1: Practice with the Diabetes Dataset

About the Diabetes Dataset

The Scikit-learn `Diabetes` dataset contains 10 physiological features (age, sex, BMI, blood pressure, etc.) from 442 patients. The goal is to predict a quantitative measure of disease progression one year later based on these features.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_diabetes


# Load data
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target
feature_names = diabetes.feature_names


df = pd.DataFrame(X, columns=feature_names)
```

Encoding Categorical Data

Problem: ML models need numbers, not text labels like "Red", "Green", or "Blue".

Two Types of Categorical Data:

1. **Ordinal:** The categories have a meaningful order.

   Example: Small < Medium < Large

2. **Nominal:** The categories have no intrinsic order.

   Example: USA, Japan, Vietnam

The encoding strategy depends on the data type.

Encoding Technique 1: OrdinalEncoder

What it does: Maps each category to an integer.

'S' -> 0, 'M' -> 1, 'L' -> 2

When to Use:

Only for Ordinal data. The numerical mapping preserves the natural order.

Warning: Using it on nominal data (e.g., 'USA' -> 0, 'Japan' -> 1) will introduce a fake order that can confuse the model.

How to use with `sklearn`

```python
from sklearn.preprocessing import OrdinalEncoder

df_sizes = pd.DataFrame({'size': ['S', 'M', 'L', 'XL', 'M', 'S']})

# Specify the desired order
size_order = ['S', 'M', 'L', 'XL']

encoder = OrdinalEncoder(categories=[size_order])
encoded_data = encoder.fit_transform(df_sizes[['size']])
```

Question: If the order is not specified in `OrdinalEncoder`, how does `sklearn` assign numbers?

Encoding Technique 2: OneHotEncoder

What it does: Creates a new binary (0/1) column for each category.

When to Use:

For Nominal data. It avoids creating a false sense of order.

Drawback: If a feature has many categories (high cardinality), it creates too many new columns (Curse of Dimensionality).

How to use with `sklearn`

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False) #
sparse_output=False returns a numpy array
encoded_data =
encoder.fit_transform(df_colors[['color']])
```

color	->	color_Red	color_Green	color_Blue
Red	->	1	0	0
Green	->	0	1	0
Blue	->	0	0	1

Question: Should I use `pd.get_dummies` or `sklearn.OneHotEncoder`?

Lab #2: Practice with the Breast Cancer Dataset

About the Breast Cancer Dataset: The Breast Cancer dataset from UCI contains characteristics of tumors extracted from images. The goal is to classify a tumor as benign (`'no-recurrence-events'`) or malignant (`'recurrence-events'`). This data is mostly categorical, making it well-suited for practicing encoding techniques.

```python
# Load data from UCI URL
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/breast-
cancer/breast-cancer.data'
columns = [
    'Class', 'age', 'menopause', 'tumor-size', 'inv-nodes',
    'node-caps', 'deg-malig', 'breast', 'breast-quad', 'irradiat'
]
df_cancer = pd.read_csv(url, header=None, names=columns, na_values='?')
```

Part II - Distribution Transforms

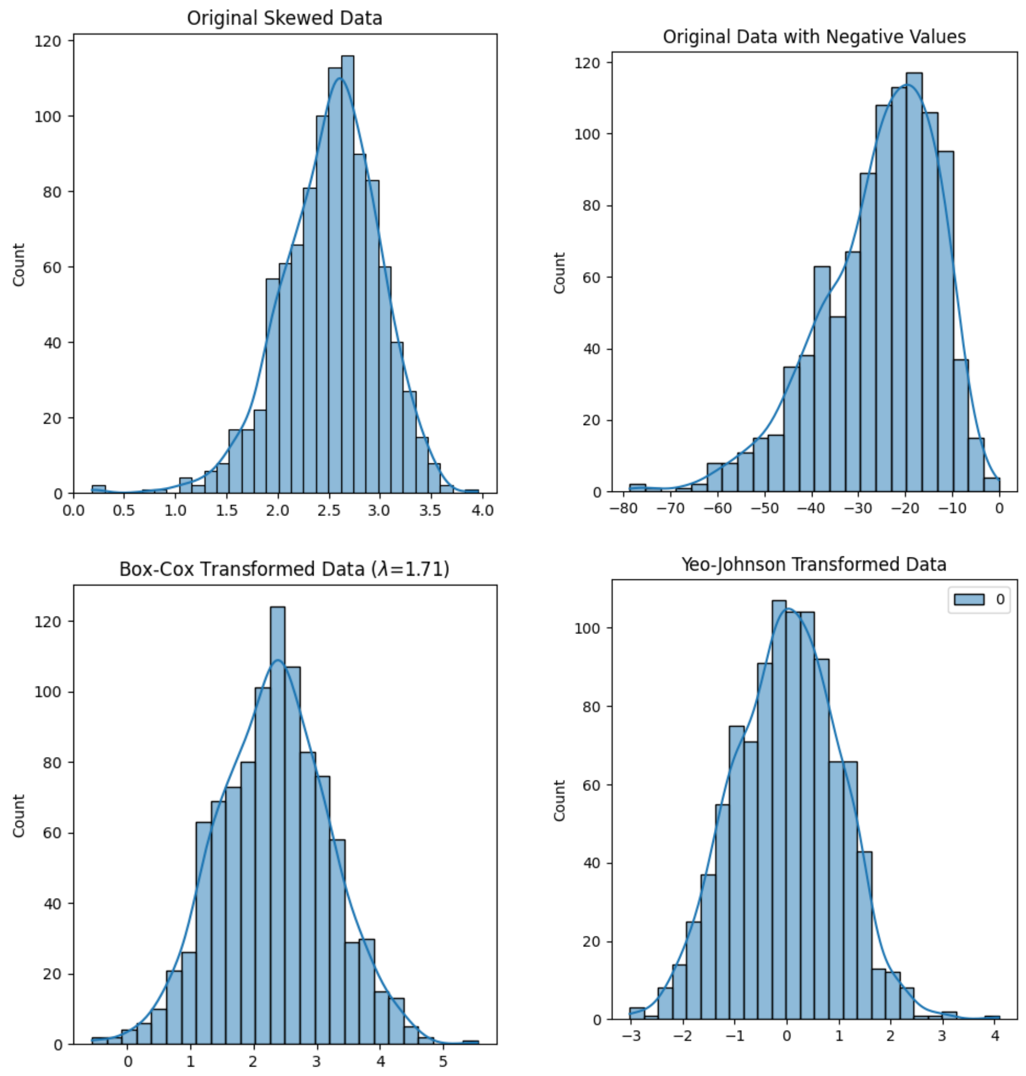
Goal: Change the *shape* of a feature's distribution.

Why?

- Many models work better if features have a **Gaussian (normal) distribution**.
- It can help stabilize variance and reduce the effect of skewed data.

Key Techniques:

- PowerTransformer (Box-Cox, Yeo-Johnson)
- QuantileTransformer



Distribution Transform 1: PowerTransformer

What it does: Applies a mathematical function (like log, square root) to make a distribution less skewed and more Gaussian.

Methods:

- **Box-Cox:** Very effective, but **only works on strictly positive data (> 0)**.

$$y^{(\lambda)} = \begin{cases} \frac{y^{\lambda}-1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(y) & \text{if } \lambda = 0 \end{cases}$$

- **Yeo-Johnson:** More flexible, works with **positive, zero, and negative data**. This is often the better default choice.

$$y^{(\lambda)} = \begin{cases} \frac{(y+1)^{\lambda}-1}{\lambda} & \text{if } \lambda \neq 0, y \geq 0 \\ \log(y+1) & \text{if } \lambda = 0, y \geq 0 \\ -\frac{(-y+1)^{2-\lambda}-1}{2-\lambda} & \text{if } \lambda \neq 2, y < 0 \\ -\log(-y+1) & \text{if } \lambda = 2, y < 0 \end{cases}$$

The algorithms **automatically finds the optimal value** of λ (usually via Maximum Likelihood Estimation) to make the distribution of $y^{(\lambda)}$ as normal as possible.

Distribution Transform 1: PowerTransformer

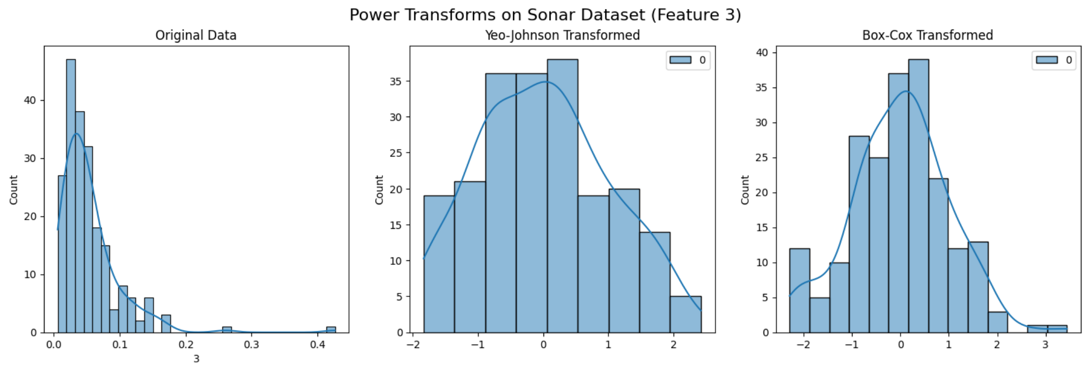
How to use `sklearn`

What?

```
from sklearn.preprocessing import PowerTransformer

# Use sklearn's PowerTransformer
pt_boxcox = PowerTransformer(method='box-cox', standardize=False)
sklearn_boxcox_data = pt_boxcox.fit_transform(skewed_data.reshape(-1, 1))

# Use PowerTransformer with method='yeo-johnson'
pt_yj = PowerTransformer(method='yeo-johnson', standardize=True)
yj_data = pt_yj.fit_transform(data_with_neg.reshape(-1, 1))
```



Question: How to handle zero or negative data with Box-Cox?

Lab 3: Practice with the Sonar Dataset

About the Sonar Dataset: The `Sonar` dataset from UCI is used to distinguish between a metal object (like a mine) and a rock underwater. The data consists of 60 energy values from sonar signals returned from different angles. All features are continuous numbers, making it suitable for practicing distribution transforms.

```
# Load Sonar data
url_sonar = 'https://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connectionist-bench/sonar/sonar.all-data'
df_sonar = pd.read_csv(url_sonar, header=None)
X_sonar = df_sonar.iloc[:, :-1]
```

Distribution Transform 2: QuantileTransformer

Definition:

- **Uniform Quantile Transform:** Transforms the data to have a uniform distribution in the range `[0, 1]`.
- **Normal Quantile Transform:** Transforms the data to have a normal (Gaussian) distribution.

When to use it?

- When the data has a very complex, unclear, or outlier-heavy distribution that Power Transforms cannot handle effectively.
- Useful for distance-based models (like KNN) as it prevents data points from clumping together.

Warning: This transformation can lose some information about the original distribution's structure.

Distribution Transform 2: QuantileTransformer

How to use `sklearn`

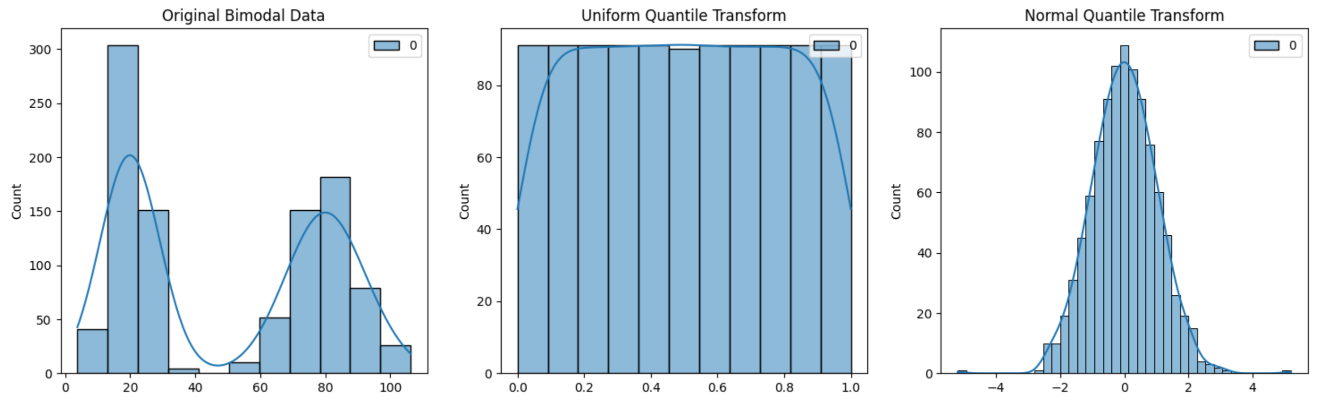
```
from sklearn.preprocessing import QuantileTransformer

# Generate bimodal data
np.random.seed(42)
d1 = np.random.normal(loc=20, scale=5, size=500)
d2 = np.random.normal(loc=80, scale=10, size=500)
bimodal_data = np.concatenate([d1, d2]).reshape(-1, 1)

# Apply QuantileTransformer
qt_uniform = QuantileTransformer(output_distribution='uniform', n_quantiles=100)
uniform_data = qt_uniform.fit_transform(bimodal_data)

qt_normal = QuantileTransformer(output_distribution='normal', n_quantiles=100)
normal_data = qt_normal.fit_transform(bimodal_data)
```

What?



Lab #4: Practice with the Sonar Dataset

Other Transforms: Discretization & Polynomial Features

Discretization (KBinsDiscretizer)

- **What:** Converts a continuous feature into categorical bins (e.g., Age 25 -> bin 20-30).
- **Why:** Can help linear models capture non-linear effects.

Polynomial Features (PolynomialFeatures)

- **What:** Creates new features by combining existing ones (e.g., from a and b, create a^2, b^2, and a*b).
- **Why:** Allows linear models to learn non-linear relationships and interactions. Crucial to scale data *before* this step.

KBinsDiscretizer

Definition: `sklearn.preprocessing.KBinsDiscretizer` is the main tool for discretization in scikit-learn. It offers 3 main strategies:

1. **‘uniform’**: All bins have equal width. E.g., if data ranges from 0-100 and `n_bins=5`, the bins would be [0, 20], (20, 40], (40, 60], (60, 80], (80, 100]. This is the simplest method.
2. **`quantile`**: All bins have the same number of data points (approximately equal frequency). E.g., if `n_bins=5`, the bins would be defined by the 0%, 20%, 40%, 60%, 80%, and 100% percentiles. This method handles skewed distributions well.
3. **`kmeans`**: Applies 1D K-Means clustering to find natural clusters in the data. Data points are assigned to the bin of the nearest cluster center. This can find more meaningful bins if the data has a clear cluster structure.

KBinsDiscretizer

- **When to use which strategy?**

- `**uniform**`: When you believe the variable's value has a linear significance.

- `**quantile**`: When you want to handle skewed distributions and ensure each bin has enough data for the model to learn from.

- `**kmeans**`: When you want bins to represent high-density clusters of data.

- **How to use `sklearn`**

```
from sklearn.preprocessing import KBinsDiscretizer

# Continuous data
continuous_data = np.random.randn(100, 1) * 10 + 50

# Initialize Discretizers with different strategies
disc_uniform = KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='uniform')
disc_quantile = KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='quantile')
disc_kmeans = KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='kmeans')
```

Question: How many `n_bins` should I choose?

Lab #5: Practice with the Sonar Dataset

PolynomialFeatures Transform

Definition: `sklearn.preprocessing.PolynomialFeatures` is the tool for creating polynomial features.

Key Parameters:

- **`degree`**: The degree of the polynomial. Higher degrees create more features.

- **`interaction_only`**: If `True`, only interaction features are created (e.g., `ab`), not power features of a single variable (e.g., `a^2`, `b^2`).

- **`include_bias`**: If `True`, adds a column of ones (the bias or intercept feature).

When to use it?

- When you suspect non-linear relationships or interactions between features.

- Often combined with linear models.

- **Warning**: A high `degree` can generate a huge number of features, leading to overfitting and increased computational cost. It's usually best to stick to degrees 2 or 3.

PolynomialFeatures Transform

How to use `sklearn`

```
from sklearn.preprocessing import PolynomialFeatures


X_simple = np.array([[2, 3], [4, 5]])


# Create 2nd degree polynomial features
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X_simple)
```

Question: Should I scale data before creating Polynomial Features?

Lab #6: Practice with the Sonar Dataset

Transforming the Target Variable (y)

- **Problem:** In regression, the target variable (e.g., house price) is often skewed. This violates assumptions of linear models and can lead to poor predictions
- **Solution:** Apply a transformation (e.g., log, sqrt) to make the target's distribution more normal.
- **CRITICAL STEP:** After the model makes a prediction, you **MUST apply the inverse transformation** to get the final result in its original units.
    - log(price) -> exp(prediction)
- **Tool:** sklearn.compose.TransformedTargetRegressor automates this process safely.

Transforming the Target Variable (y)

Common Transformations

The same transformations applied to features can also be applied to the target:

- **Log Transform** (`np.log1p`): Very effective for right-skewed, positive target variables (e.g., price, count). `log1p` (calculating $\log(1 + y)$) is preferred over `log` because it can handle zero values.
- **Box-Cox Transform**: A more powerful transformation that automatically finds the optimal λ parameter. Requires the target to be positive.
- **Yeo-Johnson Transform**: A generalized version of Box-Cox that can handle negative, zero, and positive values.

Transforming the Target Variable (y)

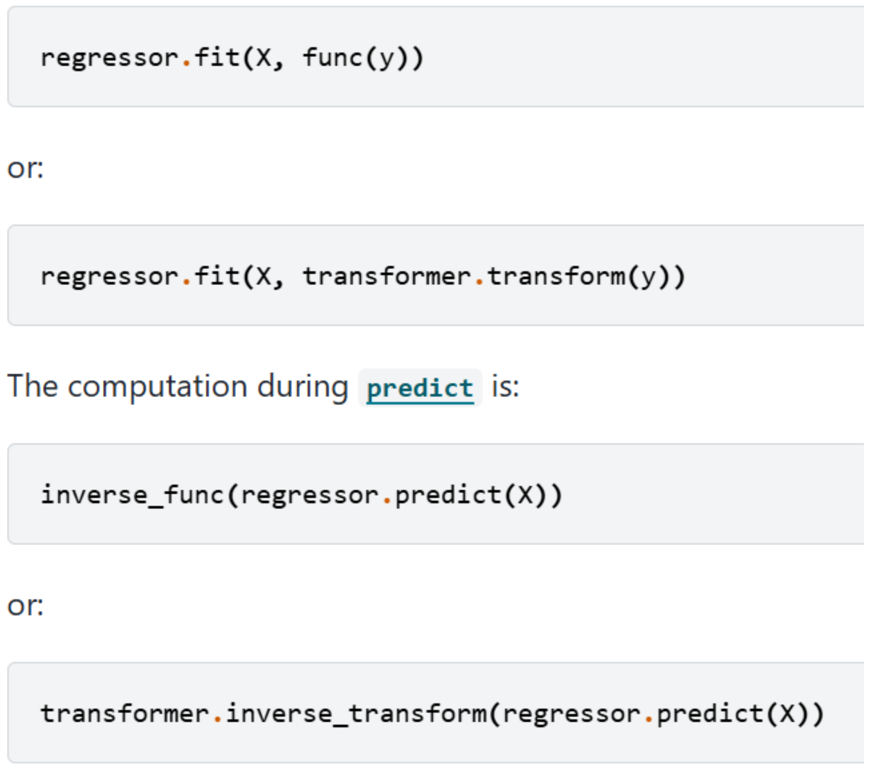
How to use with sklearn

```
from sklearn.compose import TransformedTargetRegressor
from sklearn.linear_model import LinearRegression


# Build model with TransformedTargetRegressor
# Using Log transform
ttr = TransformedTargetRegressor(
    regressor=LinearRegression(),
    func=np.log1p,          # The transformation function
    inverse_func=np.expm1   # The inverse transformation
function
)

# Train the model
ttr.fit(X_train, y_train)

# Get predictions (already inverse-transformed
automatically)
y_pred = ttr.predict(X_test)
```



```
regressor.fit(X, func(y))
```

or:

```
regressor.fit(X, transformer.transform(y))
```

The computation during `predict` is:

```
inverse_func(regressor.predict(X))
```

or:

```
transformer.inverse_transform(regressor.predict(X))
```

Summary & Key Takeaways

1. **Data transformation is not optional; it's essential.**
2. **Scale** numerical data to a common range (StandardScaler is a great default).
3. **Encode** categorical data appropriately (OneHotEncoder for nominal, OrdinalEncoder for ordinal).
4. Consider transforming the **distribution** of skewed features and targets (PowerTransformer).

Q&A