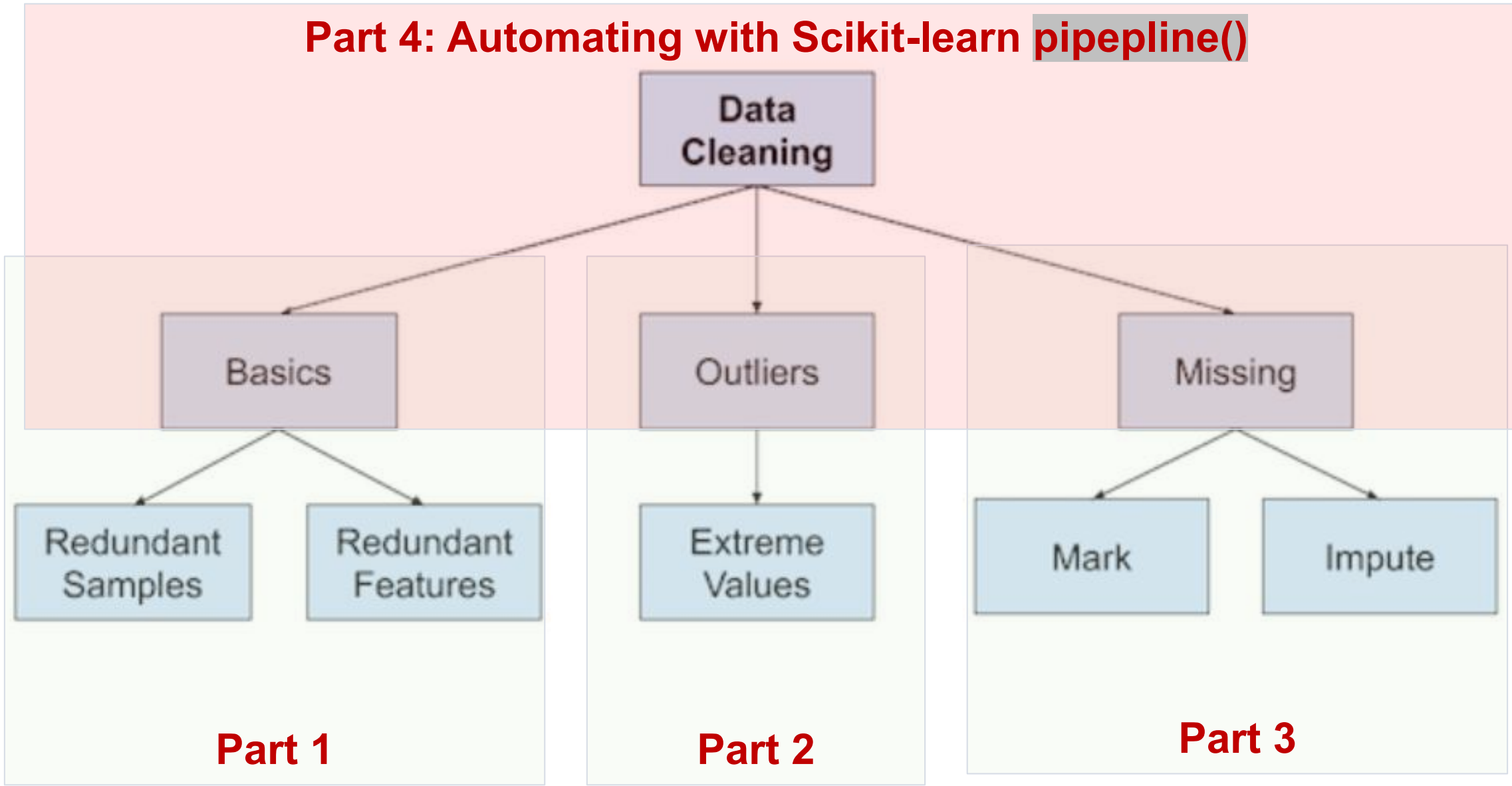
Lesson 6: Data Cleaning

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

Overview Data Cleaning



CONTENTS

What We Will Cover Today

Why is Data Cleaning Crucial?

"Garbage In, Garbage Out"

- Machine learning models learn patterns from the data they are given.
- Flawed, noisy, or incorrect data leads to unreliable and inaccurate models.
- **Goal of Data Cleaning:** To create a high-quality, reliable dataset that serves as a solid foundation for model training.

Impact of Clean Data:

- Improved model performance and accuracy.
- More robust and generalizable models.
- Prevents errors during model training.

Part 1 - Basic Data Cleaning

Foundational Steps for Data Integrity

- **Removing Zero-Variance Features:** Features that have the same value for all samples.
- **Removing Duplicate Rows:** Identical observations that can bias the model.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import io
```

Data set

We will use a simplified version of the `oil-spill` dataset for illustration.

```
# Create sample oil-spill data
csv_data = '''f_1,f_2,f_3,f_4,f_5
1,25.4,3.8,0,10
2,22.3,4.1,0,12
3,26.1,3.7,0,10
4,24.8,3.9,0,11
2,22.3,4.1,0,12''' # Duplicate row


df_oil = pd.read_csv(io.StringIO(csv_data))
print("Initial oil-spill data:")
print(df_oil)
```

Zero-Variance Features

Concept:

- A feature where all values are identical has a variance of 0.

$$Var(X) = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}$$

- These features provide no information for the model to learn from.

Tool for the Job:

- **Scikit-learn Class:** sklearn.feature_selection.VarianceThreshold
- A simple transformer that removes all features with variance below a certain threshold.

Practice with VarianceThreshold

Usage:

- Initialize the transformer: transformer = VarianceThreshold(threshold=0)
- Apply to data: data_cleaned = transformer.fit_transform(data)

```
# Import the required class
from sklearn.feature_selection import VarianceThreshold
```

```
# Use VarianceThreshold to remove columns with zero variance
transformer = VarianceThreshold(threshold=0)
# Note: VarianceThreshold only works on numerical data
data_transformed = transformer.fit_transform(df_oil)

# Get the names of the retained columns
retained_cols = transformer.get_feature_names_out(input_features=df_oil.columns)

# Create a new DataFrame
data_cleaned = pd.DataFrame(data_transformed, columns=retained_cols)

print(data_cleaned)
```

Constant Categorical Features: nunique

Concept:

- The same principle applies to categorical features. A column where every entry is the same category (e.g., 'USA') offers no predictive value.

Tool for the Job:

- **Pandas Method:** pandas.DataFrame.nunique()
- This method counts the number of unique values in each column.
- **Usage:**
  - Identify columns where df.nunique() == 1.
  - Drop these columns using df.drop().

```
df_oil.drop(columns=df_oil.columns[df_oil.nunique()==1])
```

Duplicate Rows: Concept & Tool

Concept:

- Rows that are exact copies of each other.
- **Risks:** Can lead to data leakage and cause the model to overweight certain patterns.

Tool for the Job:

- **Pandas Method:** `pandas.DataFrame.drop_duplicates()`
- A straightforward method to identify and remove duplicate rows.

```python
# Check for duplicate rows
print(f"\nNumber of duplicate rows:
{df_oil.duplicated().sum()}")

# Remove duplicate rows
data_no_dup = df_oil.drop_duplicates()
print("\nData after removing duplicate rows:")
print(data_no_dup)
```

Lab #1: Practice with full data

- The data information: link
- Removing Zero-Variance Features
- Removing Duplicate Rows.

```
url_lab1 = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/oil-
spill.csv’

df_lab1 = pd.read_csv(url_lab1, header=None)
df_lab1.head()
```

Part 2 - Handling Outliers

What are Outliers?

- Observations that are significantly different from other observations.
- **Causes:** Measurement errors, data entry mistakes, or genuinely rare events.
- **Impact:** Can skew statistical measures and disproportionately influence model parameters.

Data set

We will use the `housing` dataset for illustration. This dataset contains information about house prices, and extremely large or small values could be outliers.

```
# Create sample housing data
csv_housing = '''CRIM,ZN,INDUS,CHAS,NOX,RM,AGE,DIS,RAD,TAX,PTRATIO,B,LSTAT,MEDV
0.00632,18,2.31,0,0.538,6.575,65.2,4.09,1,296,15.3,396.9,4.98,24
0.02731,0,7.07,0,0.469,6.421,78.9,4.9671,2,242,17.8,396.9,9.14,21.6
0.02729,0,7.07,0,0.469,7.185,61.1,4.9671,2,242,17.8,392.83,4.03,34.7
0.03237,0,2.18,0,0.458,6.998,45.8,6.0622,3,222,18.7,394.63,2.94,33.4
0.06905,0,2.18,0,0.458,7.147,54.2,6.0622,3,222,18.7,396.9,5.33,36.2
0.9,80,20,0,0.6,12,90,2,5,666,20,350,30,500''' # Row that may contain outliers


df_housing = pd.read_csv(io.StringIO(csv_housing))
print("Initial housing data:")
df_housing
```

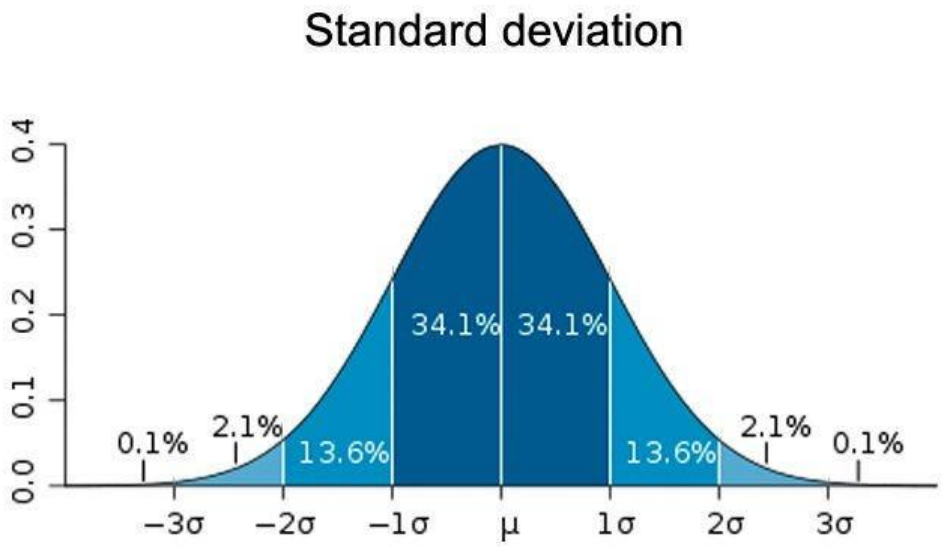
Outlier Detection: Standard Deviation Method

Concept:

- Assumes data follows a Gaussian (normal) distribution.
- Identifies data points outside a specified number of standard deviations (σ) from the mean (μ).
- A common rule is to use 3 standard deviations.

When to Use:

- When your data is normally or near-normally distributed.



% of values expected to lie in the symmetric interval (−zσ, zσ)

1σ	68.2689492%
2σ	95.4499736%
3σ	99.7300204%
4σ	99.993666%
5σ	99.9999426697%

Practice with Standard Deviation

Detect outliers in one column

- Upper bound: $\mu + 2 \times \sigma$
- Lower bound: $\mu - 2 \times \sigma$

```
# Consider the MEDV column (house price)
data_col = df_housing['MEDV']

# Calculate the limits
mean, std = data_col.mean(), data_col.std()
cut_off = std * 2 # Using 2 std to make outliers more visible in this small dataset
lower, upper = mean - cut_off, mean + cut_off

# Identify outliers
outliers = df_housing[(data_col < lower) | (data_col > upper)]
print(f"Found {len(outliers)} outliers.")
print(outliers[['RM', 'MEDV']])

# Remove outliers
data_cleaned_outlier = df_housing[(data_col >= lower) & (data_col <= upper)]
print(f"\nOriginal data size: {len(df_housing)}")
print(f"Data size after cleaning: {len(data_cleaned_outlier)}")
```

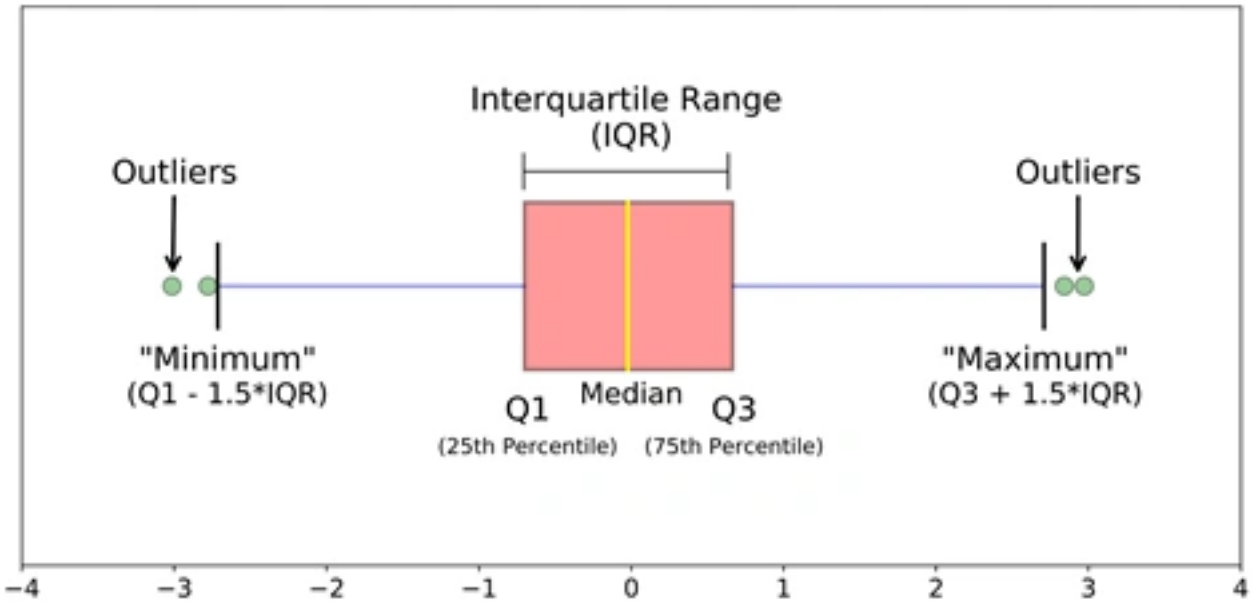
Outlier Detection: Interquartile Range (IQR) Method

Concept:

- A robust method that does not assume a specific data distribution.
- Uses quartiles to define boundaries.
  - **IQR:** $Q3\ (75th\ percentile)\ -\ Q1\ (25th\ percentile)$
  - **Upper bound:** $Q3\ +\ 1.5\ \times IQR$
  - **Lower bound:** $Q1\ -\ 1.5\ \times IQR$

When to Use:

- For skewed data or when you need a method that is less sensitive to extreme values.



Practice with IQR


```
# Reusing the MEDV column
data_col = df_housing['MEDV']


# Calculate Q1, Q3, and IQR
Q1 = data_col.quantile(0.25)
Q3 = data_col.quantile(0.75)
IQR = Q3 - Q1


# Calculate the limits
lower_iqr, upper_iqr = Q1 - 1.5 * IQR, Q3 + 1.5 * IQR


# Identify outliers
outliers_iqr = df_housing[(data_col < lower_iqr) | (data_col > upper_iqr)]
print(f"Found {len(outliers_iqr)} outliers using IQR.")
print(outliers_iqr[['RM', 'MEDV']])
```

Outlier Detection: Local Outlier Factor (LOF)

Concept:

- An unsupervised, model-based approach that identifies outliers based on local density.
- Excellent for identifying outliers in a multivariate (multi-feature) setting.

Tool for the Job:

- **Scikit-learn Class:** `sklearn.neighbors.LocalOutlierFactor`
- Returns 1 for inliers and -1 for outliers.
- **Correct Train/Test Usage (Important!)**
  - **On Training Data:** Use `.fit_predict()` to learn the data distribution and identify outliers. These outliers should then be removed: `yhat_train = lof.fit_predict(X_train)`
  - **On Test Data:** Use only .predict(). **Do not** fit again and **do not** remove outliers from the test set: `yhat_test = lof.predict(X_test)`
  - **Warning:** Never fit on the combined dataset before splitting. This causes data leakage.

Practice with LocalOutlierFactor


```
# Import the required class
from sklearn.neighbors import LocalOutlierFactor
```

Important Parameters:

- `n_neighbors` (int, default=20): The number of neighbors used to calculate the local density. This is the most important parameter to tune.
- `contamination` (float, default='auto'): The expected proportion of outliers in the dataset (e.g., 0.1 for 10%). This parameter affects the model's decision threshold. The default 'auto' will determine the threshold based on the original algorithm's publication.

```
# Use LOF on the entire housing dataset
lof = LocalOutlierFactor()
yhat = lof.fit_predict(df_housing)

# Filter out the outliers (LOF labels outliers as -1)
mask = yhat != -1
print(f"Number of outliers found: {sum(yhat == -1)}")
print(f"Data size after removing outliers: {df_housing[mask].shape}")
```

Lab #2: Practice with full data set

Data set information: link

```
column_names = [
    "CRIM",     # Tội phạm bình quân đầu người theo thị trấn
    "ZN",       # Tỷ lệ đất ở > 25,000 sq.ft
    "INDUS",    # Tỷ lệ diện tích cho doanh nghiệp phi bán lẻ
    "CHAS",     # Biến giả sông Charles (=1 nếu gần sông, 0 nếu không)
    "NOX",      # Nồng độ oxit nitơ (phần triệu)
    "RM",       # Số phòng trung bình mỗi căn hộ
    "AGE",      # % căn hộ xây dựng trước 1940
    "DIS",      # Khoảng cách bình quân đến 5 trung tâm việc làm
    "RAD",      # Chỉ số khả năng tiếp cận đường cao tốc
    "TAX",      # Thuế bất động sản
    "PTRATIO", # Tỷ lệ học sinh/giáo viên
    "B",        # 1000(Bk - 0.63)^2, với Bk % dân da đen
    "LSTAT",    # % dân có địa vị kinh tế xã hội thấp
    "MEDV"      # Giá trị trung vị của nhà (ngàn USD)
]
url_lab2 = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv"
df_lab2 = pd.read_csv(url_lab2, header=None, names=column_names)
df_lab2
df_lab2.head()
```

The Challenge of Missing Values

- Most machine learning algorithms cannot handle missing values (NaN).
- The way we handle them can significantly impact model performance.
- We will explore four main strategies:
  1. Removing Data
  2. Statistical Imputation
  3. KNN Imputation
  4. Iterative Imputation

Data set

We will use a simplified version of the `horse-colic` dataset for illustration.

```
# Create sample horse-colic data
csv_horse = '''hospital_number,rectal_temp,pulse,respiratory_rate,pain,outcome
530101,38.5,66,28,3,2
534817,39.2,88,20,?,1
530334,38.3,40,?,3,1
529048,39.1,164,84,4,2
526254,?,72,?,2,1'''


# Read data, considering '?' as a missing value
df_horse = pd.read_csv(io.StringIO(csv_horse), na_values='?')
print("Initial horse-colic data:")
print(df_horse)
```

Missing Data: Removing Rows/Columns

Concept:

- The simplest strategy: delete any row containing missing values.
- **Tool:** pandas.DataFrame.dropna()

Pros & Cons:

- **Pros:** Quick and easy.
- **Cons:** Can result in significant data loss and may introduce bias.

```
# Check the number of missing values
print("\nNumber of missing values per column:")
print(df_horse.isnull().sum())


# Remove rows with missing values
data_dropped = df_horse.dropna()
print("\nData after dropping missing rows:")
print(data_dropped)
```

Missing Data: Statistical Imputation

Concept:

- Replace missing values with a statistical summary of the column (mean, median, mode).

Tool for the Job:

- **Scikit-learn Class:** sklearn.impute.SimpleImputer
- **Usage:**
  - Initialize with a strategy: `imputer = SimpleImputer(strategy='mean')`
  - Apply to data: `data_imputed = imputer.fit_transform(data)`

Practice with SimpleImputer


```
# Import the required class
from sklearn.impute import SimpleImputer
```

Important Parameters:

- `strategy` (string, default='mean'): The imputation strategy. Possible values are `'mean'`, `'median'`, `'most_frequent'`, or `'constant'`.
- `fill_value` (string or number, default=None): When `strategy='constant'`, this parameter is used to specify the value to be imputed.

```
# Use SimpleImputer
imputer = SimpleImputer(strategy='mean')
data_imputed_mean = imputer.fit_transform(df_horse)


print("Data after imputing with the mean:")
print(pd.DataFrame(data_imputed_mean, columns=df_horse.columns))
```

Missing Data: K-Nearest Neighbors (KNN) Imputation

Concept:

- A multivariate approach that finds the k most similar rows (neighbors).
- Imputes the missing value using the average value from those neighbors.

Tool for the Job:

- **Scikit-learn Class:** sklearn.impute.KNNImputer
- **Usage:**
  - Initialize with number of neighbors: imputer = KNNImputer(n_neighbors=5)
  - Apply to data: data_imputed = imputer.fit_transform(data)

Practice with KNNImputer


```
# Import the required class
from sklearn.impute import KNNImputer
```

Important Parameters:

- `n_neighbors` (int, default=5): The number of neighbors to use for imputation.
- `weights` (string, default='uniform'): The weight function used in prediction. `'uniform'` means all neighbors are weighted equally. `'distance'` means that closer neighbors will have a greater influence.

```
# Use KNNImputer
knn_imputer = KNNImputer(n_neighbors=2, weights='uniform')
data_imputed_knn = knn_imputer.fit_transform(df_horse)


print("Data after KNN imputation:")
print(pd.DataFrame(data_imputed_knn, columns=df_horse.columns))
```

Missing Data: Iterative Imputation (MICE)

Concept:

- An advanced strategy that treats imputation as a machine learning problem.
- It models each feature with missing values as a function of all other features.

Tool for the Job:

- **Scikit-learn Class:** `sklearn.impute.IterativeImputer` (Experimental)
- **Usage:**
  - Initialize the imputer: `imputer = IterativeImputer()`
  - Apply to data: `data_imputed = imputer.fit_transform(data)`

Practice with IterativeImputer


```
# Import the required classes
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
```

Important Parameters:

- `estimator` (object, default=BayesianRidge()): The regression model used to predict missing values. Other models like `RandomForestRegressor` can be used.
- `max_iter` (int, default=10): The maximum number of imputation rounds.
- `random_state` (int): To ensure reproducible results.

```
# Use IterativeImputer
iter_imputer = IterativeImputer(max_iter=10, random_state=0)
data_imputed_iter = iter_imputer.fit_transform(df_horse)


print("Data after iterative imputation:")
print(pd.DataFrame(data_imputed_iter, columns=df_horse.columns))
```

Lab #3: Practice with full data: horse-colic

The data information: link

```
url_lab3 = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/horse-colic.csv"

col_names = [
    "surgery", "age", "hospital_number", "rectal_temp", "pulse",
    "respiratory_rate", "temp_extremities", "peripheral_pulse",
    "mucous_membrane", "capillary_refill", "pain", "peristalsis",
    "abdominal_distension", "nasogastric_tube", "nasogastric_reflux",
    "nasogastric_reflux_ph", "rectal_exam_feces", "abdomen",
    "packed_cell_volume", "total_protein", "abdomocentesis_appearance",
    "abdomocentesis_total_protein", "outcome", "surgical_lesion",
    "lesion_1", "lesion_2", "lesion_3", "lesion_4"
]

df_lab3 = pd.read_csv(url_lab3, header=None, names=col_names, na_values="?")

df_lab3.head()
```

Part 4 - Automating with Pipelines

The Danger of Data Leakage

- **Data Leakage:** When information from outside the training dataset is used to create the model.
- **Example:** Calculating the mean for imputation using the *entire* dataset before splitting into train/test sets.
- This leads to overly optimistic performance metrics and models that fail in production.

The Solution: Scikit-learn Pipelines

What is a Pipeline?

- **Tool:** sklearn.pipeline.Pipeline
- It chains together multiple steps (e.g., an imputer and a classifier) into a single object.
- When you call .fit(), it correctly fits the transformers *only on the training data*.
- **Usage:**
  - Define steps as a list of tuples:

    steps = [('imputer', SimpleImputer()), ('model', RandomForestClassifier())]
  - Create the pipeline: pipeline = Pipeline(steps)
  - Use like a normal model: pipeline.fit(X_train, y_train)

Details on Components within the Pipeline

Let's break down the tools we're using:

- **sklearn.model_selection.train_test_split**
  - **Purpose:** A crucial function that splits your dataset into two subsets: one for training the model and one for testing its performance on unseen data.
  - **Why?** This prevents the model from "memorizing" the data and ensures an honest evaluation of its ability to generalize.
- **sklearn.impute.SimpleImputer**
  - **Purpose:** As we've learned, this transformer handles missing (NaN) values.
  - **Role in Pipeline:** It's the first step, ensuring the data is complete before it's passed to the model.
- **sklearn.ensemble.RandomForestClassifier**
  - **Purpose:** A powerful classification model that builds multiple decision trees and merges their outputs for a more accurate prediction.
  - **Role in Pipeline:** It's the final step, the estimator that learns from the preprocessed data.

Practice with Pipeline


```
# Import all necessary classes for this example
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer # Already imported, but good practice to have it here
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

```
# 1. Prepare data from horse-colic
# Drop rows where the target variable 'outcome' is missing
df_lab3_clean = df_lab3.dropna(subset=['outcome'])
X = df_lab3_clean.drop('outcome', axis=1)
y = df_lab3_clean['outcome']

# 2. Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# 3. Create the Pipeline
# This Pipeline will consist of 2 steps:
# 'imputer': Impute missing values with the median.
# 'model': Train a Random Forest model.
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('model', RandomForestClassifier(random_state=42))
])
```

```
# 4. Train the entire Pipeline on the training set
# Scikit-learn will automatically:
# - Call imputer.fit_transform(X_train)
# - Then use the result to train model.fit(X_train_transformed, y_train)
pipeline.fit(X_train, y_train)


# 5. Evaluate the Pipeline on the test set
# Scikit-learn will automatically:
# - Call imputer.transform(X_test) (NOT re-fitting)
# - Then use the result to predict model.predict(X_test_transformed)
y_pred = pipeline.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)


print(f"Pipeline has been trained.")
print(f"Accuracy on the test set: {accuracy:.4f}")
```

Summary and Key Takeaways

1. **Start with the Basics:** Always remove zero-variance features and duplicates first.
2. **Choose the Right Outlier Method:** Use IQR for robustness, or model-based methods like LOF for multivariate data.
3. **Select an Imputation Strategy:** Progress from SimpleImputer to KNNImputer or IterativeImputer as needed.
4. **ALWAYS Use Pipelines:** Encapsulate your preprocessing and modeling steps to prevent data leakage and create production-ready code

Q&A