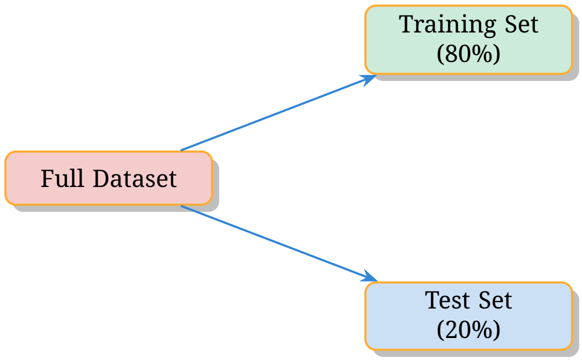
Chapter 2: ML Pipeline — A Visual Feast!

Chapter 2 of *Hands-On Machine Learning* turns raw data into a model-ready masterpiece, like crafting a gourmet dish from scratch! We've organized Scikit-Learn's key tools into a logical flow: splitting and stratifying data, cleaning, transforming, feature engineering, and modeling. Each tool tackles a specific problem, shown with a colorful diagram and highlighted code. A master pipeline diagram ties it all together. Let's dive in!

1. train_test_split

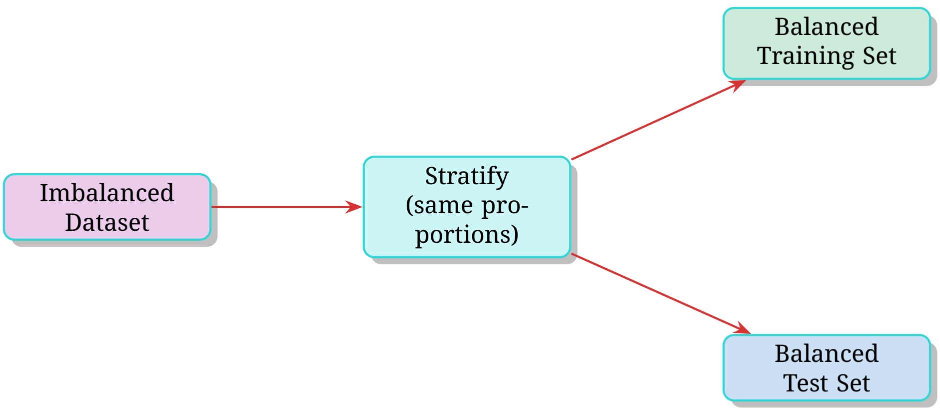
Problem: Models need unseen data to evaluate generalization, but random splits can imbalance classes.
What it does: Splits data into training and test sets, optionally with stratification.
Key Settings: `test_size=0.2`, `random_state=42`, `stratify=y` (for balanced classes).
Key Methods: Returns `X_train, X_test, y_train, y_test`.




```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

2. Stratifying Data

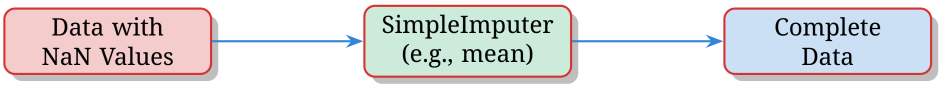
Problem: Imbalanced classes (e.g., 90% class A, 10% class B) in splits can bias model training.
What it does: Ensures training and test sets have the same class proportions as the original dataset using `stratify=y` in `train_test_split`.
Key Settings: `stratify=y` in `train_test_split`.
Key Methods: Applied within `train_test_split`.




```
from sklearn.model_selection import train_test_split
# Ensure class proportions are maintained
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

3. SimpleImputer

Problem: Missing values (NaN) break models or skew results.
What it does: Fills missing data with a statistic or constant.
Key Settings: `strategy="mean"`, `"median"`, `"most_frequent"`, or `"constant"` (with `fill_value`).
Key Methods: `fit(X)`, `transform(X)`, `fit_transform(X)`.




```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="mean")
imputer.fit(X_train)  # Learn mean
X_filled = imputer.transform(X_train)  # Fill NaNs
```

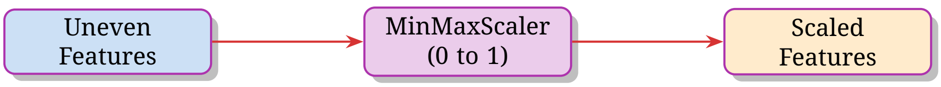
4. StandardScaler

Problem: Features on different scales confuse models like linear regression.
What it does: Scales features to mean=0, std=1.
Key Settings: `with_mean=True`, `with_std=True`.
Key Methods: `fit(X)`, `transform(X)`, `fit_transform(X)`.


```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)  # Learn mean/std
X_scaled = scaler.transform(X_train)  # Scale data
```

5. MinMaxScaler

Problem: Some models need features in a specific range (e.g., [0, 1]).
What it does: Scales features to a fixed range, typically [0, 1].
Key Settings: `feature_range=(0, 1)`.
Key Methods: `fit(X)`, `transform(X)`, `fit_transform(X)`.




```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
scaler.fit(X_train)  # Learn min/max
X_scaled = scaler.transform(X_train)  # Scale to [0, 1]
```

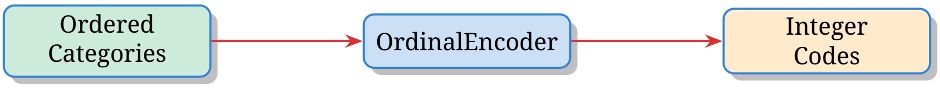
6. OneHotEncoder

Problem: Models can't process categorical data (e.g., colors) directly.
What it does: Converts categories to binary columns.
Key Settings: `sparse_output=False`, `handle_unknown="ignore"`, `drop="first"`.
Key Methods: `fit(X)`, `transform(X)`, `fit_transform(X)`.


```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse_output=False, drop="first")
encoder.fit(X_train)  # Learn categories
X_encoded = encoder.transform(X_train)  # Create binary columns
```

7. OrdinalEncoder

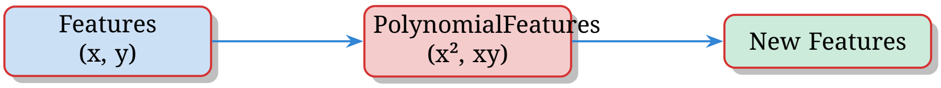
Problem: Ordered categories (e.g., low, high) need numerical encoding.
What it does: Assigns integers to ordered categories.
Key Settings: `handle_unknown="use_encoded_value"`.
Key Methods: `fit(X)`, `transform(X)`, `fit_transform(X)`.




```
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder()
encoder.fit(X_train)  # Learn category order
X_encoded = encoder.transform(X_train)  # Assign numbers
```

8. PolynomialFeatures

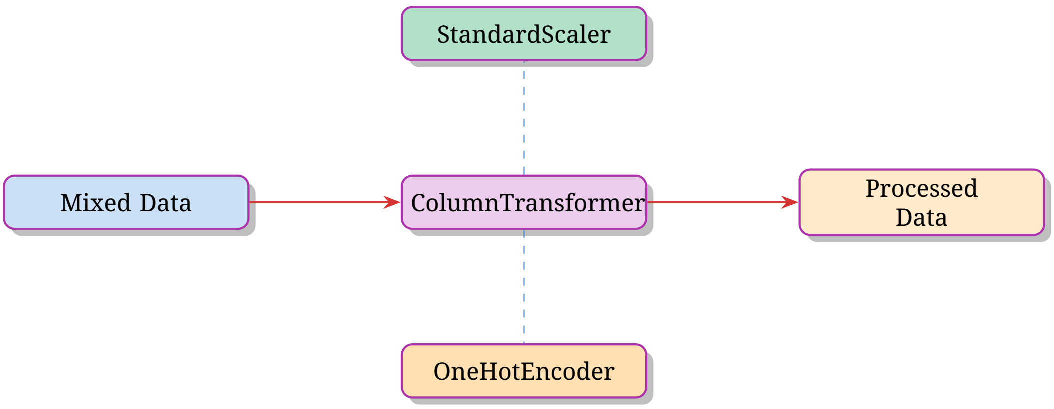
Problem: Linear models can't capture non-linear patterns (e.g., quadratic relationships).
What it does: Generates polynomial and interaction terms.
Key Settings: `degree=2`, `include_bias=True`.
Key Methods: `fit(X)`, `transform(X)`, `fit_transform(X)`.




```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=False)
poly.fit(X_train)  # Learn polynomial terms
X_poly = poly.transform(X_train)  # Add polynomial features
```

9. ColumnTransformer

Problem: Mixed data types need different preprocessing steps.
What it does: Applies specific transformers to selected columns.
Key Settings: `transformers=[("name", transformer, columns), ...]`, `remainder="passthrough"`.
Key Methods: `fit(X)`, `transform(X)`, `fit_transform(X)`.




```
from sklearn.compose import ColumnTransformer
preprocessor = ColumnTransformer([
    ("num", StandardScaler(), [0, 1]),
    ("cat", OneHotEncoder(sparse_output=False), [2])
])
preprocessor.fit(X_train)  # Fit transformers
X_processed = preprocessor.transform(X_train)  # Apply
```

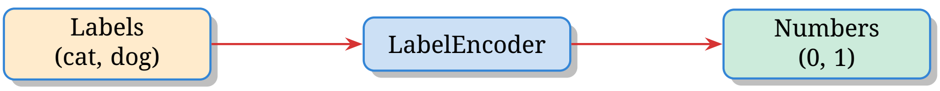
10. Pipeline

Problem: Manual preprocessing is error-prone and repetitive.
What it does: Chains preprocessing and modeling into one workflow.
Key Settings: `steps=[("name", transformer), ...]`.
Key Methods: `fit(X, y)`, `predict(X)`.


```python
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="mean")),
    ("preprocessor", preprocessor),
    ("model", LinearRegression())
])
pipeline.fit(X_train, y_train)  # Fit all steps
y_pred = pipeline.predict(X_test)  # Transform and predict
```

11. LabelEncoder

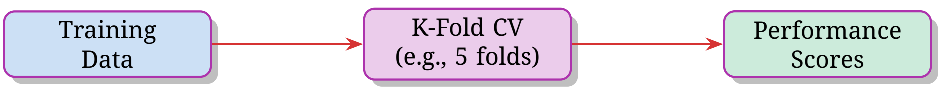
Problem: Classification models require numerical target labels.
What it does: Converts labels (e.g., "cat", "dog") to numbers.
Key Methods: `fit(y)`, `transform(y)`, `fit_transform(y)`, `inverse_transform(y)`.




```python
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y_train)  # Encode labels
y_original = encoder.inverse_transform(y_encoded)  # Decode
```

12. Cross-Validation (K-Fold)

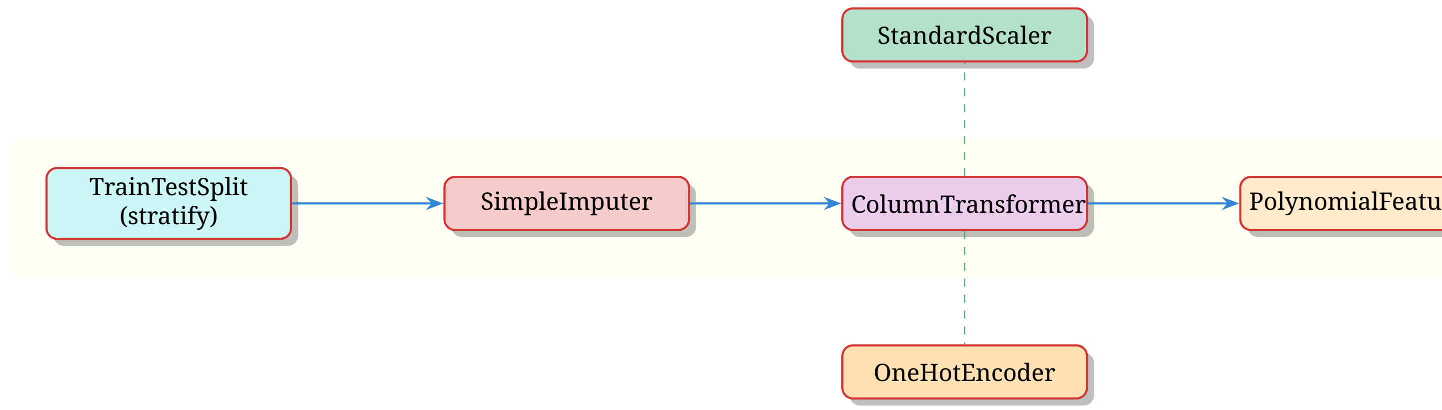
Problem: A single train-test split may not reliably assess model performance.
What it does: Splits training data into k folds, trains on k-1, tests on 1, repeats k times.
Key Settings: `cv=k` (e.g., 5), `scoring="accuracy"` (or other metric).
Key Methods: `cross_val_score(estimator, X, y)` returns scores.




```python
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
model = LinearRegression()
scores = cross_val_score(model, X_train, y_train, cv=5, scoring="neg_mean_squared_error")
```

13. Master Pipeline

Problem: Combining all steps manually is complex and risks errors.
What it does: Orchestrates the full workflow from splitting to modeling.



figureMaster ML pipeline: From stratified splitting to modeling, all steps flow seamlessly.

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder, PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Split with stratification
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42,
     stratify=y)

# Define preprocessor
preprocessor = ColumnTransformer([
    ("num", StandardScaler(), [0, 1]),
    ("cat", OneHotEncoder(sparse_output=False), [2])
])

# Build pipeline
pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="mean")),
    ("preprocessor", preprocessor),
    ("poly", PolynomialFeatures(degree=2, include_bias=False)),
    ("model", LinearRegression())
])

# Fit and predict
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
```