Lesson 9: The Comprehensive ML Workflow with Scikit-learn Pipelines

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

Agenda

1. **The Foundation:** `train_test_split`
2. **The Basic** `Pipeline`
3. **The Problem:** Handling Mixed Data Types - `ColumnTransformer`
4. **The Full Workflow:** `Pipeline`
5. **Reliable Evaluation:** `KFold` & `cross_val_score`
6. **Deployment:** Saving & Loading Pipelines with `joblib`
7. **Summary** & Key Takeaways

Sample Data

	age	income	city	education	gender	target
0	25.0	50000.0	Hanoi	Bachelor	Male	0
1	30.0	60000.0	HCMC	Master	Female	1
2	45.0	100000.0	Hanoi	PhD	Male	1
3	55.0	80000.0	Danang	Master	Female	0
4	NaN	120000.0	HCMC	Bachelor	Male	1

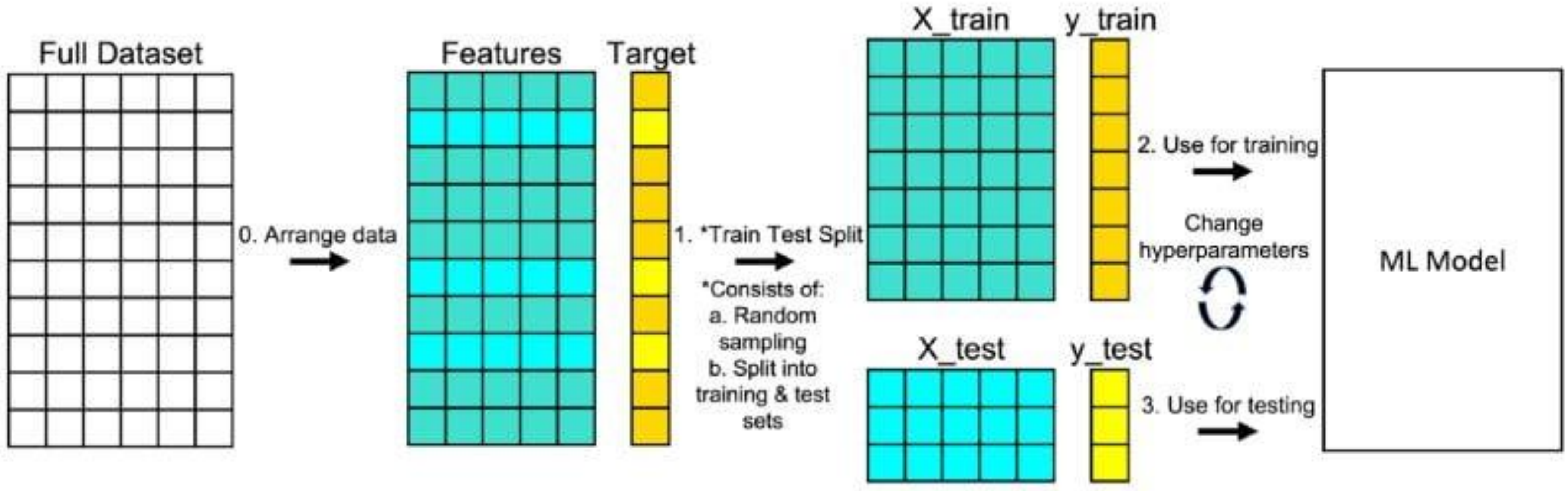
```
# --- Create San
# We create a cc                                                    ic columns,
categorical colu
data = {
    'age': [25, 30, 45, 55, np.nan, 35, 60, 65, 70, 22, 48, 52],
    'income': [50000, 60000, 100000, 80000, 120000, 75000, np.nan, 200000,
180000, 45000, 90000, 110000],
    'city': ['Hanoi', 'HCMC', 'Hanoi', 'Danang', 'HCMC', 'Danang', 'Hanoi',
'HCMC', 'Hanoi', 'Danang', 'HCMC', 'Hanoi'],
    'education': ['Bachelor', 'Master', 'PhD', 'Master', 'Bachelor', 'PhD',
'Master', 'PhD', 'Bachelor', 'Bachelor', 'Master', 'PhD'],
    'gender': ['Male', 'Female', 'Male', 'Female', 'Male', 'Female', 'Male',
'Female', 'Male', 'Male', 'Female', 'Female'],
    'target': [0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1]
}
df = pd.DataFrame(data)
```

Part 1: The Foundation: `train_test_split`

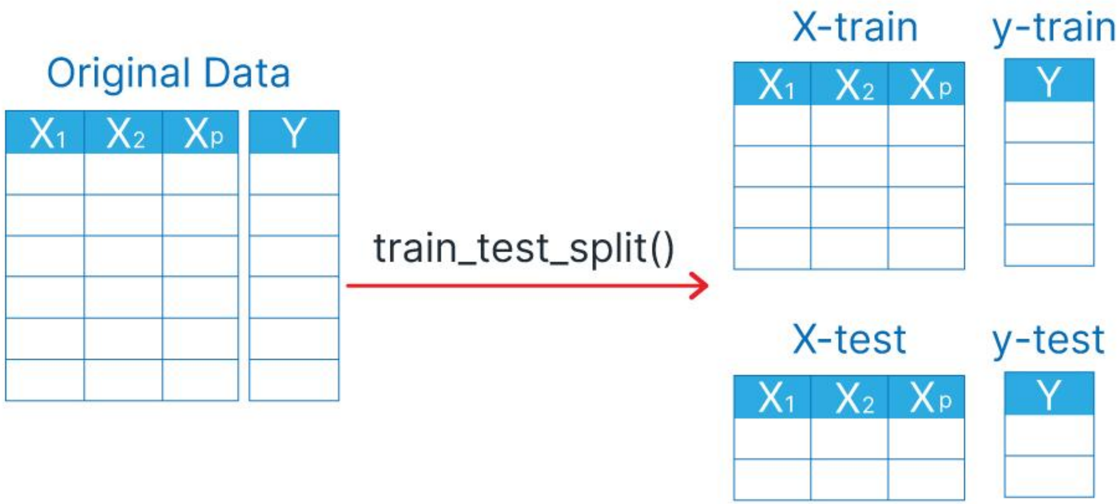
Why?

It's the **mandatory first step**. We must isolate a Test Set to prevent **Data Leakage**.

- All "learning" (`.fit()`) happens *only* on the training data.
- The test set is "unseen data" used *only* for final evaluation.



Part 1: The Foundation: `train_test_split`



Key Parameters

```python
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,     # e.g., 20% of data for testing
    random_state=42,   # For reproducible results
    stratify=y         # Keeps class balance (CRITICAL for classification)
)
```

Question #1

1. **Why must we `train_test_split` *before* any preprocessing (like scaling or imputation)?**
   To prevent Data Leakage from the test set into the training process.
2. **What is the purpose of the `stratify=y` parameter?**
   To ensure the class proportions (e.g., % of 0s and 1s) in `y_train` and `y_test` are the same as the original `y`.
3. **What happens if you forget to set `random_state`?**
   You will get a different split every time you run the code, making your results not reproducible.

Part 2: The Basic Pipeline

Before we handle complex data, let's understand *why* we use a Pipeline.

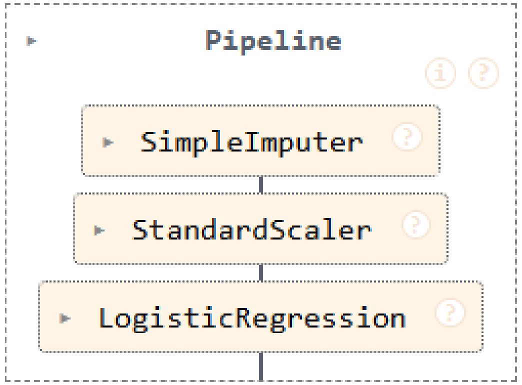
Think of it as an **assembly line**. For numerical data, the steps are:

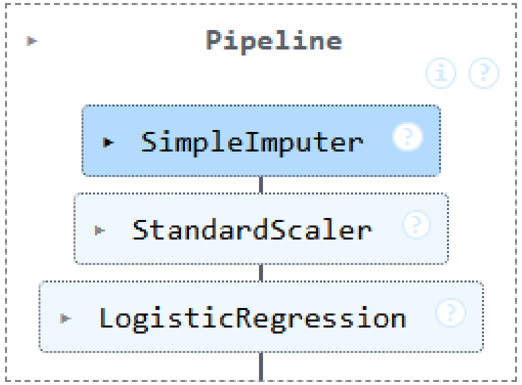
1. Fill missing values (Impute)
2. Scale features
3. Train model

A Pipeline bundles these steps into one object.

The Basic Pipeline

```python
# 1. Define the steps for the simple pipeline
# This pipeline only works on numerical data
simple_numerical_steps = [
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(random_state=42))
]

# 2. Create the simple Pipeline object
simple_num_pipeline =
Pipeline(steps=simple_numerical_steps)

# Select only the numerical features for this example
numeric_features = ['age', 'income'] # From Part 1

# 3. Fit the pipeline on the numerical training data
simple_num_pipeline.fit(X_train[numeric_features], y_train)

# 4. Score the pipeline on the numerical test data
score = simple_num_pipeline.score(X_test[numeric_features], y_test)
```





Question 2

1. **What is the main benefit of using a Pipeline here?**

   It automates the process and prevents data leakage by correctly calling fit_transform on train data and only transform on test data.

2. **What is the required format for the steps parameter?**

   A list of (name, object) tuples, like ('imputer', SimpleImputer(...)).

3. **What would happen if we ran simple_pipe.fit(X_train, y_train) (using the *full* X_train)?**

   It would crash. The StandardScaler (step 2) would get string columns like 'city' and fail.

Part 3: Handling Mixed Data Types

Problem: Real-world data is messy.

	age	income	city	education	gender	target
0	25.0	50000.0	Hanoi	Bachelor	Male	0
1	30.0	60000.0	HCMC	Master	Female	1
2	45.0	100000.0	Hanoi	PhD	Male	1
3	55.0	80000.0	Danang	Master	Female	0
4	NaN	120000.0	HCMC	Bachelor	Male	1

A single dataset can have:

- **Numerical Features:** age, income
  - *Needs:* Imputation (filling missing values), Scaling
- **Nominal Features:** city, gender (No order)
  - *Needs:* Imputation, One-Hot Encoding
- **Ordinal Features:** education ('Bachelor', 'Master', 'PhD')
  - *Needs:* Imputation, Ordinal Encoding

How can we apply different steps to different columns easily?

Question 3

1. **What is the key difference between a Nominal and an Ordinal feature?**

   A Nominal feature has no inherent order (e.g., city). An Ordinal feature has a meaningful order (e.g., education: 'Bachelor' < 'Master').

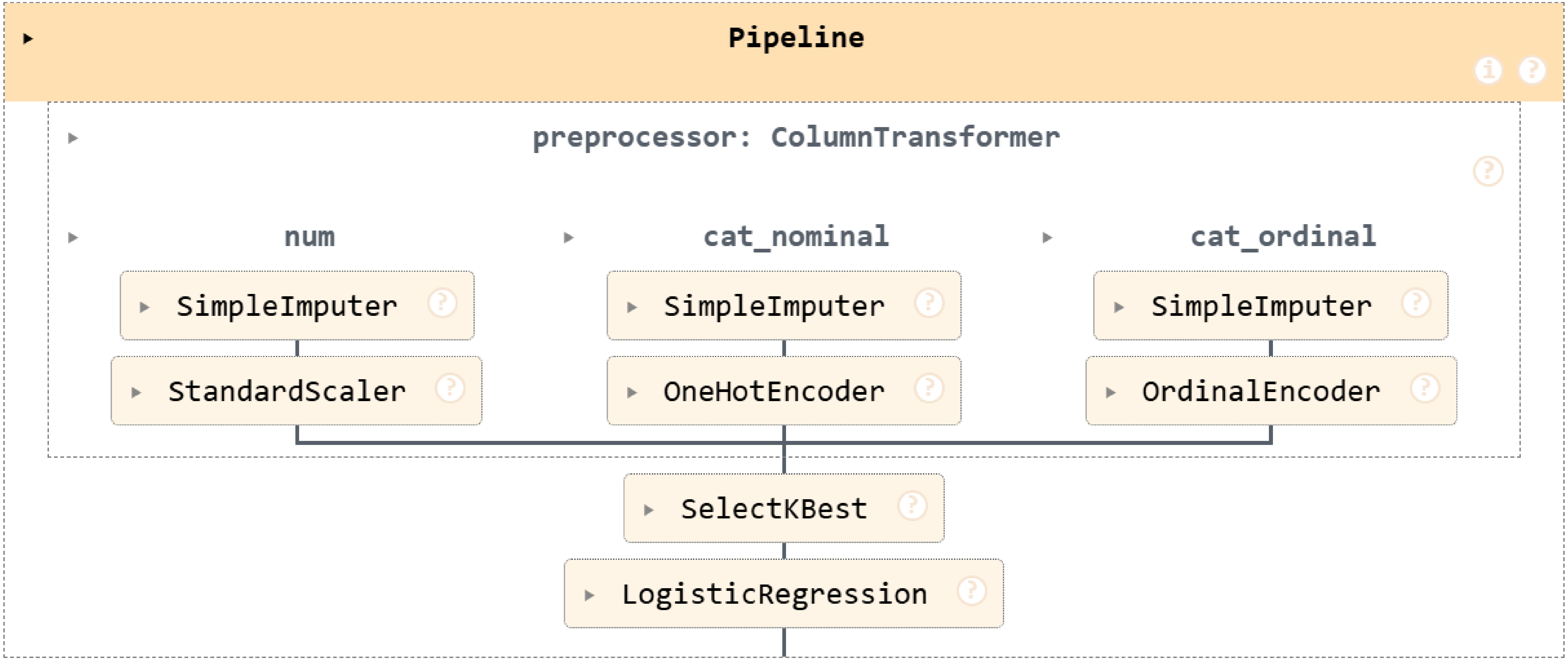
2. **Why can't we just use OrdinalEncoder (e.g., 0, 1, 2) for the city column?**

   It would create a fake mathematical relationship (e.g., 'Danang' (2) > 'Hanoi' (0)), which can confuse the model.

	age	income	city	education	gender	target
0	25.0	50000.0	Hanoi	Bachelor	Male	0
1	30.0	60000.0	HCMC	Master	Female	1
2	45.0	100000.0	Hanoi	PhD	Male	1
3	55.0	80000.0	Danang	Master	Female	0
4	NaN	120000.0	HCMC	Bachelor	Male	1

The Solution: ColumnTransformer

ColumnTransformer applies different **sub-pipelines** to different **columns** in parallel.



The Solution: ColumnTransformer

ColumnTransformer applies different **sub-pipelines** to different **columns** in parallel.

Step 1: Define Sub-Pipelines

```
# 1.1 Create a sub-pipeline for NUMERICAL data
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

# 1.2 Create a sub-pipeline for CATEGORICAL (NOMINAL) data
nominal_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# 1.3 CREATE SUB-PIPELINE FOR CATEGORICAL (ORDINAL) DATA - CHANGE
education_order = ['Bachelor', 'Master', 'PhD']

ordinal_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('ordinal', OrdinalEncoder(categories=[education_order],
                               handle_unknown='use_encoded_value',
                               unknown_value=-1))
])
```

The Solution: ColumnTransformer

Step 2: Combine Sub-Pipelines: We tell the `ColumnTransformer` which pipeline to apply to which columns.

```
# 2.1 Define column lists
numeric_features = ['age', 'income']
nominal_features = ['city', 'gender'] # Nominal columns (no order)
ordinal_features = ['education']     # Ordinal columns (has order)

# 2.2 Combine with ColumnTransformer
# ColumnTransformer takes a list of 'transformers'
# Each transformer is a tuple: (name, sub_pipeline, list_of_columns_to_apply_to)
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat_nominal', nominal_transformer, nominal_features), # Pipeline for nominal columns
        ('cat_ordinal', ordinal_transformer, ordinal_features)  # Pipeline for ordinal columns
    ])
```

Question 4

1. **What is the main purpose of the `ColumnTransformer`?**
   To apply different transformation pipelines to different subsets of columns in parallel.
2. **What does the `handle_unknown='ignore'` parameter in `OneHotEncoder` do?**
   It prevents an error if the model sees a new category in the test data (e.g., a new city) that it never saw in the training data.
3. **Why do we put `SimpleImputer` *inside* the sub-pipelines?**
   To prevent data leakage. This way, the `median` (for numbers) and `most_frequent` (for categories) are learned *only* from the training folds.
4. **What happens to features that are *not* listed in any of the `transformers`? And how would you keep them?**
   By default, they are dropped (`remainder='drop'`). To keep them, you must set `remainder='passthrough'` in the `ColumnTransformer`.

Part 4: The Full Workflow: The Advanced Pipeline

Now we chain everything together:

1. **Preprocessor:** Clean & Transform (The ColumnTransformer we just built)
2. **Selector:** Feature Selection (e.g., SelectKBest)
3. **Model:** The final estimator (e.g., LogisticRegression)

```
full_pipeline = Pipeline(steps=[
    # STEP 1: Cleaning + Transform (Using the ColumnTransformer)
    ('preprocessor', preprocessor),

    # STEP 2: Feature Selection (Select features)
    # SelectKBest: Select 'k' best features
    # score_func=f_classif: Use f_classif (ANOVA F-test) to score features
    # (After processing, we have 2 numeric + 5 OHE (city+gender) + 1 ordinal = 8 features)
    ('selector', SelectKBest(score_func=f_classif, k=6)), # Select 6 of the 8 best features

    # STEP 3: Modeling
    # LogisticRegression: The final model for prediction
    ('model', LogisticRegression(random_state=42))
])
```

Question 4

1. **What is the purpose of the full_pipeline object?**

   To chain all steps of the ML workflow (preprocessing, selection, modeling) into a single object that can be fit and predict with.

2. **Does the order of steps in the Pipeline matter?**

   Yes, absolutely. Data must be cleaned/transformed *before* features can be selected, and features must be selected *before* the model is trained.

3. **What does the ('selector', SelectKBest(k=6)) step do?**

   It selects the top 6 features that have the strongest relationship with the target variable, based on the f_classif (ANOVA F-test) score.

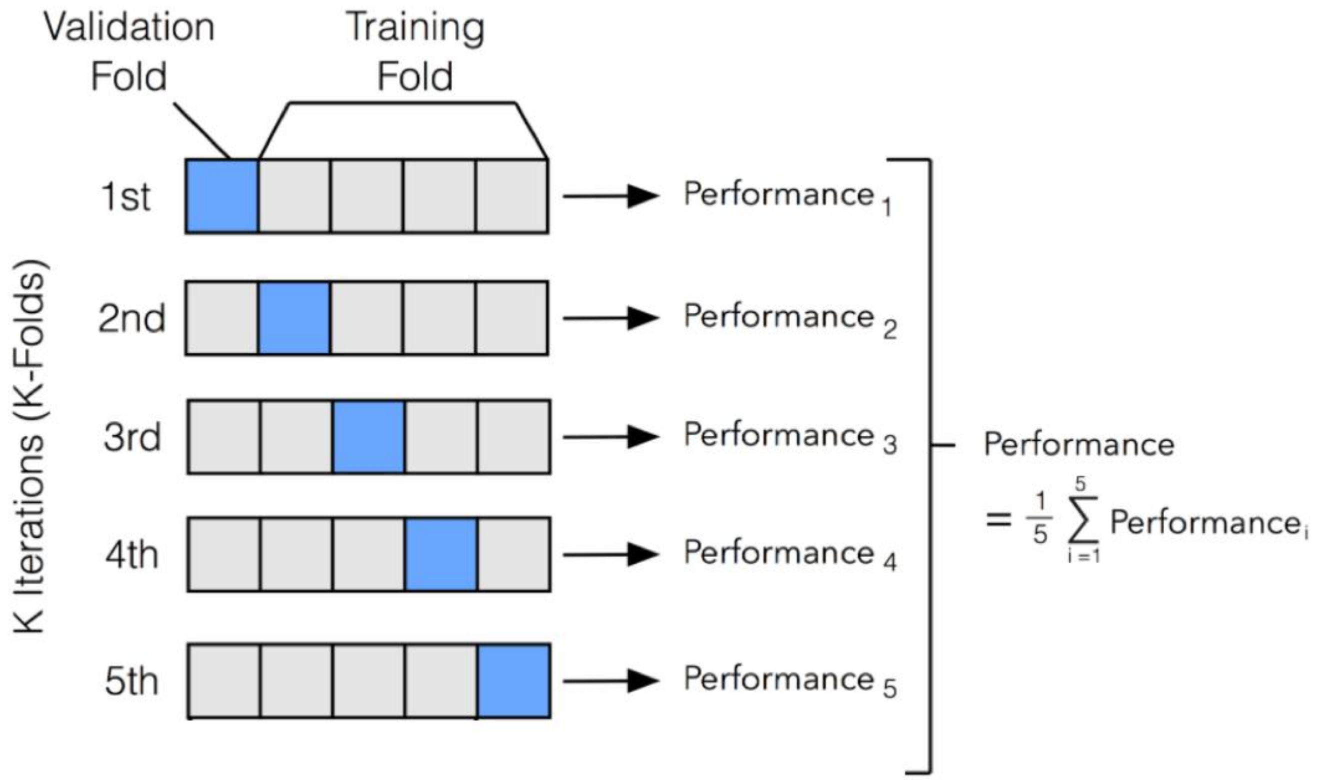
4. **(Advanced)** What if I want to use f_classif on numeric features and chi2 on categorical features?

   You can't do this with a single SelectKBest *after* the preprocessor. The solution is to remove the ('selector', ...) step from the main Pipeline and **move the SelectKBest steps *inside* the sub-pipelines** (e.g., add SelectKBest(f_classif) to numeric_transformer and SelectKBest(chi2) to nominal_transformer).

Part 5: Reliable Evaluation: Cross-Validation

A single train_test_split score can be lucky or unlucky. We need a more robust method.

- **Kfold:** Splits data into k parts ("folds"). Trains on k-1 folds and tests on 1 fold, repeating k times.
- **cross_val_score:** Automates the K-Fold process.



Cross-Validation


```
# We use cross-validation on the entire original X and y
# (Or X_train, y_train if you want to tune parameters on the train set)
# Here, we use (X, y) to get the most general evaluation


# Define the K-Fold splitting strategy (e.g., 5 folds)
kfold = KFold(n_splits=5, shuffle=True, random_state=42)


# Call cross_val_score with the ENTIRE pipeline
cv_scores = cross_val_score(full_pipeline, X, y,
                            cv=kfold,
                            scoring='accuracy')


print(f"Cross-Validation Scores (5-fold): {cv_scores}")
print(f"Mean Accuracy: {cv_scores.mean():.4f}")
print(f"Standard Deviation: {cv_scores.std():.4f}")
```

Question 5

1. **Why is cross_val_score (e.g., with 5 folds) generally better than a single train_test_split for evaluating a model?**

   A single split might be 'lucky' or 'unlucky'. CV gives a more stable and reliable estimate of model performance by averaging 5 different splits.

2. **What is the correct object to pass into cross_val_score's estimator argument?**

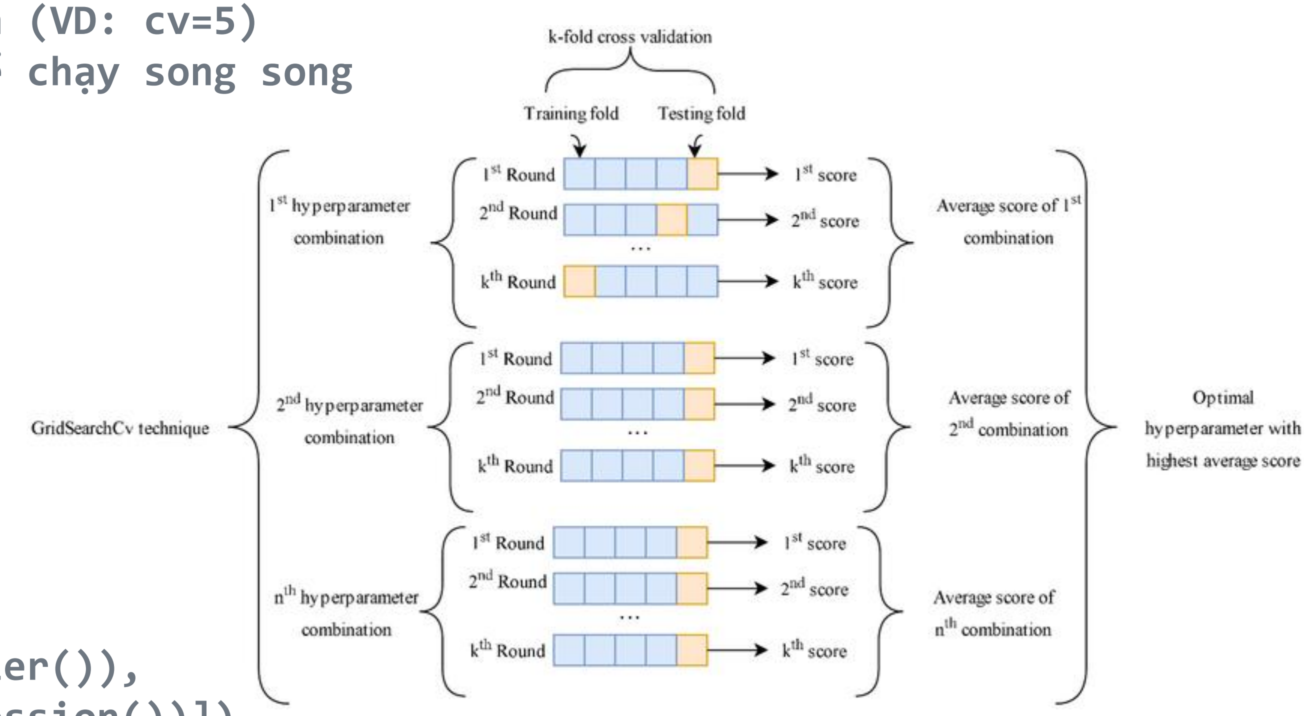
   The full_pipeline. This is critical to prevent data leakage during cross-validation.

3. **What does a high Standard Deviation from cv_scores tell us?**

   It means the model's performance was very inconsistent across different folds, suggesting the model is unstable.

Tuning the hyper parameters - GridsearchCV

```
grid = GridSearchCV(
    estimator,          # Mô hình bạn muốn tinh chỉnh (ex: SVC, RandomForestClassifier, pipeline ...)
    param_grid,         # Tập hợp các tham số để thử
    scoring=None,       # Chỉ số đánh giá (accuracy, f1, r2,...)
    cv=None,            # Số lần cross-validation (VD: cv=5)
    n_jobs=-1,          # Dùng tất cả CPU core để chạy song song
    verbose=1           # Hiển thị tiến trình)


# estimator = SVC()
param_grid = {
    'C': [0.1, 1, 10], # Tên tham số 'C'
    'kernel': ['linear', 'rbf'],
    'gamma': [0.01, 0.001]
}


# estimator = Pipeline([('scaler', StandardScaler()),
#                       ('model', LogisticRegression())])
param_grid = {
    'scaler__with_std': [True, False], # Tham số with_std của bước 'scaler'
    'model__C': [0.1, 1.0, 10.0],       # Tham số C của bước 'model'
    'model__penalty': ['l2’]            # Tham số penalty của bước 'model'
}
```



Part 6: Deployment - Saving & Loading Pipelines

When you're ready to deploy, you don't just save the model... you save the **ENTIRE PIPELINE**.

This ensures new data is preprocessed in the *exact same way* as the training data.

Save the Pipeline

```
import joblib

# Use joblib.dump to 'freeze' the entire pipeline
(including imputer, scaler, model...)
model_filename = 'final_model_pipeline.joblib'
joblib.dump(full_pipeline, model_filename)
```

Load the pipeline

```
# Use joblib.load to restore the saved pipeline

loaded_pipeline = joblib.load(model_filename)
print("Pipeline loaded successfully.")
```

Question 6

1. **Why is it better to save the full_pipeline object instead of just the model object?**

   The pipeline contains all preprocessing steps (imputer, scaler, encoders). Saving it ensures new data is processed *exactly* the same way as the training data, preventing errors.

2. **What function from joblib is used to save a pipeline? What function is used to load it?**

   joblib.dump() to save, joblib.load() to load.

3. **What must be true about the X_new data frame used for prediction?**

   It must have the *exact same* column names and structure as the original X_train data (even if it contains missing values).

Summary

1. **train_test_split** is the mandatory first step.
2. **Pipeline** bundles steps and prevents data leakage (start with a simple one).
3. **ColumnTransformer** is the key to handling mixed data types (numerical, nominal, ordinal).
4. **Pipeline (Advanced)** chains the ColumnTransformer with SelectKBest and a model.
5. **cross_val_score(pipeline, ...)** is the correct way to evaluate your entire workflow reliably.
6. **joblib.dump(pipeline, ...)** saves the *entire workflow*, not just the model, for production.

Q&A