## **Guide: Using the Preprocessing Module for Model Training**

**Purpose:**

This module provides standardized functions for loading, splitting, and preprocessing the student dataset. Using this module ensures consistency across different models developed for the project's two main prediction tasks:

1. **First Year Persistence** (Classification)
2. **Second Term GPA** (Regression)

**Prerequisites:**

* **Libraries:** Ensure you have pandas, numpy, and scikit-learn installed. imblearn is needed if using SMOTE.
* **Data:** The module expects the student\_data.csv file to be located at ../Data/student\_data.csv relative to where the script is run.
* **Initial Setup:** The provided code currently loads the data into a global DataFrame df when the script is run. It also handles converting '?' to NaN.
  + *(Note: For better module design, consider wrapping the data loading part into its own function within the module).*
  + *(Assumption: The code assumes you have handled the Prev Education = 0.0 values before splitting, e.g., by replacing them with np.nan as discussed).*

**Core Functions:**

**1. split\_data\_for\_task(predict\_persistence, test\_size=0.2, random\_state=89)**

* **What it does:** Splits the loaded data (df) into training and testing sets (X\_train, X\_test, y\_train, y\_test).
* **Why do this?** We need separate sets to train the model and then evaluate its performance on unseen data fairly.
* **Parameters:**
  + predict\_persistence (bool): **Crucial.** Set to True if your target is 'First Year Persistence'. Set to False if your target is 'Second Term Gpa'. This determines which column is y and enables stratification for the persistence task.
  + test\_size (float): Proportion of data reserved for the test set (default is 0.2 or 20%).
  + random\_state (int): Ensures the split is the same every time (for reproducibility).
* **Returns:** A tuple: (X\_train, X\_test, y\_train, y\_test).

**2. create\_preprocessor(predict\_persistence)**

* **What it does:** Defines and returns the scikit-learn ColumnTransformer structure (the preprocessing pipeline blueprint) tailored for the specified task. It includes steps for imputation, encoding, and scaling, automatically selecting the correct input features based on the task.
* **Why do this?** Ensures consistent and correct application of preprocessing steps (like imputation based on training data only, scaling, encoding) needed to convert raw features into a format suitable for machine learning models. Handles differences in feature sets between the two tasks.
* **Parameters:**
  + predict\_persistence (bool): Set to True for the persistence task, False for the GPA task. This flag adjusts which columns are processed (e.g., excludes 'Second Term Gpa' as a feature when predicting persistence).
* **Returns:** An *unfitted* ColumnTransformer object.

**Workflow: How to Use the Module**

Here’s the standard workflow for preparing data to train *any* model:

**Step 1: Load Data**

* Run the initial part of the script to load the data into the df DataFrame.

**Step 2: Split Data for Your Task**

* Decide which task you are working on (Persistence or GPA).
* Call split\_data\_for\_task with the correct predict\_persistence flag.

# Example for Persistence Task  
X\_train, X\_test, y\_train, y\_test = split\_data\_for\_task(predict\_persistence=True, random\_state=89)  
  
# OR  
  
# Example for GPA Task  
# X\_train, X\_test, y\_train, y\_test = split\_data\_for\_task(predict\_persistence=False, random\_state=89)

**Step 3: Create and Fit the Preprocessor**

* Call create\_preprocessor with the *same* predict\_persistence flag used for splitting. This gives you the pipeline structure.
* **Fit** this preprocessor **only** on the training data (X\_train).

# Get the pipeline structure for the task  
preprocessor = create\_preprocessor(predict\_persistence=True) # Match the flag used in split  
  
# Fit the preprocessor on the training features  
print("Fitting preprocessor...")  
fitted\_preprocessor = preprocessor.fit(X\_train)  
print("Preprocessor fitted.")

**Step 4: Transform Your Data**

* Use the fitted\_preprocessor to transform both X\_train and X\_test.

print("Transforming data...")  
X\_train\_processed = fitted\_preprocessor.transform(X\_train)  
X\_test\_processed = fitted\_preprocessor.transform(X\_test)  
print("Data transformed.")  
print(f"Processed training data shape: {X\_train\_processed.shape}")  
print(f"Processed test data shape: {X\_test\_processed.shape}")

**Step 5: Handle Class Imbalance (Persistence Task Only)**

* If predict\_persistence was True, apply SMOTE (or another resampling technique) to the *processed training data only*.

# ONLY if predict\_persistence was True:  
from imblearn.over\_sampling import SMOTE # Make sure imblearn is installed  
  
print("Applying SMOTE to training data...")  
smote = SMOTE(random\_state=111) # Use a fixed random state  
X\_train\_final, y\_train\_final = smote.fit\_resample(X\_train\_processed, y\_train)  
print(f"Resampled training data shape: {X\_train\_final.shape}")  
  
# If predict\_persistence was False, just use the processed data directly:  
# X\_train\_final, y\_train\_final = X\_train\_processed, y\_train

**Step 6: Train Your Model**

* Your data (X\_train\_final, y\_train\_final, X\_test\_processed, y\_test) is now ready!
* Use X\_train\_final and y\_train\_final to fit your chosen model.
* Use X\_test\_processed and y\_test to evaluate it.

**Model Training Examples:**

*(These are minimal examples assuming you have the processed data from Step 5)*

*(Please do not use this code, they are for example only)*

**A. TensorFlow Neural Network:**

# import tensorflow as tf  
# from tensorflow import keras  
#  
# # Define your NN architecture (example)  
# model = keras.Sequential([  
# keras.layers.Dense(64, activation='relu', input\_shape=(X\_train\_final.shape[1],)),  
# keras.layers.Dense(32, activation='relu'),  
# # Output layer depends on task:  
# # keras.layers.Dense(1, activation='sigmoid') # For Persistence (binary classification)  
# # keras.layers.Dense(1) # For GPA (regression)  
# ])  
#  
# # Compile the model (choose loss based on task)  
# # model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy']) # For Persistence  
# # model.compile(optimizer='adam', loss='mean\_squared\_error') # For GPA  
#  
# print("Training TensorFlow NN...")  
# history = model.fit(X\_train\_final, y\_train\_final, epochs=50, batch\_size=32, validation\_split=0.2) # Use internal validation split  
# print("Training complete.")  
#  
# # Evaluate on test set  
# # loss, accuracy = model.evaluate(X\_test\_processed, y\_test) # For Persistence  
# # mse = model.evaluate(X\_test\_processed, y\_test) # For GPA

**B. Logistic Regression (for Persistence):**

# from sklearn.linear\_model import LogisticRegression  
# from sklearn.metrics import classification\_report  
#  
# print("Training Logistic Regression...")  
# log\_reg = LogisticRegression(max\_iter=1000, random\_state=89)  
# log\_reg.fit(X\_train\_final, y\_train\_final) # Use resampled data if predict\_persistence=True  
# print("Training complete.")  
#  
# # Evaluate  
# # y\_pred = log\_reg.predict(X\_test\_processed)  
# # print(classification\_report(y\_test, y\_pred))

**C. Linear Regression (for GPA):**

# from sklearn.linear\_model import LinearRegression  
# from sklearn.metrics import mean\_squared\_error  
#  
# print("Training Linear Regression...")  
# lin\_reg = LinearRegression()  
# lin\_reg.fit(X\_train\_final, y\_train\_final) # Use original processed data (X\_train\_processed)  
# print("Training complete.")  
#  
# # Evaluate  
# # y\_pred = lin\_reg.predict(X\_test\_processed)  
# # mse = mean\_squared\_error(y\_test, y\_pred)  
# # print(f"Test MSE: {mse}")

**D. Support Vector Machine (SVC for Persistence / SVR for GPA):**

# from sklearn.svm import SVC, SVR  
# from sklearn.metrics import classification\_report, mean\_squared\_error  
#  
# # For Persistence:  
# # print("Training SVC...")  
# # svm\_clf = SVC(random\_state=89)  
# # svm\_clf.fit(X\_train\_final, y\_train\_final) # Use resampled data  
# # print("Training complete.")  
# # y\_pred = svm\_clf.predict(X\_test\_processed)  
# # print(classification\_report(y\_test, y\_pred))  
#  
# # For GPA:  
# # print("Training SVR...")  
# # svm\_reg = SVR()  
# # svm\_reg.fit(X\_train\_final, y\_train\_final) # Use original processed data  
# # print("Training complete.")  
# # y\_pred = svm\_reg.predict(X\_test\_processed)  
# # mse = mean\_squared\_error(y\_test, y\_pred)  
# # print(f"Test MSE: {mse}")

**Step 7: Save Your Fitted Preprocessor!**

* **Why do this?** After you've trained your final model, you need to save the *exact* preprocessor that was fitted on the training data used for that model. This is essential for processing new, real-world data consistently before making predictions with your saved model.
* **How:** Use the joblib library.

import joblib  
  
# Assume 'fitted\_preprocessor' is the variable holding the pipeline fitted in Step 3  
preprocessor\_filename = 'final\_preprocessor.joblib' # Choose a descriptive name  
  
try:  
 joblib.dump(fitted\_preprocessor, preprocessor\_filename)  
 print(f"Fitted preprocessor saved successfully to {preprocessor\_filename}")  
except Exception as e:  
 print(f"Error saving preprocessor: {e}")  
  
# Later, when you need to process new data:  
# loaded\_preprocessor = joblib.load(preprocessor\_filename)  
# new\_data\_processed = loaded\_preprocessor.transform(new\_raw\_data)

**Conclusion:**

Following these steps ensures that data is loaded, split, and preprocessed consistently and correctly for each prediction task. Using the provided functions helps maintain clean, reusable code and prevents common errors like data leakage. Remember to save your fitted preprocessor alongside your trained model!