The given housing datasets contain various inconsistencies, missing values, and categorical variables that require preprocessing before they can be used for analysis. The beginning of the code notebook presents a comprehensive data preprocessing pipeline designed to address these challenges and prepare a dataset for house categorization and regression analysis.

The data preprocessing pipeline begins with dropping the 'Id' column, which serves as a unique identifier and does not contribute to the analysis. Next, we define a function called 'categorize\_house,' which assigns a specific category to each house based on its architectural style, building type, and construction year. This function enables us to create a new column, 'House Category,' which serves as the target variable for categorization analysis. Following that, we dummify the categorical target variable using one-hot encoding, resulting in binary columns representing different house categories.

To handle missing values, we calculate the count and proportion of missing values for each column in the dataset. Columns with missing values count greater than 50% are identified and subsequently dropped from the dataset. This ensures that columns with substantial missing data do not introduce bias or affect the analysis.

To preprocess the numerical features, we apply a pipeline that includes imputing missing values with the mean and scaling the features using the StandardScaler. For categorical features with high cardinality, we employ a pipeline that imputes missing values with the most frequent value and applies one-hot encoding. Similarly, ordinal categorical features are imputed with the most frequent value and encoded using the OrdinalEncoder.

We utilize the ColumnTransformer to apply different transformations to different columns based on their data types. This enables us to handle numerical, high cardinality categorical, and ordinal categorical features separately. The preprocessor is then fitted on the dataset to learn the necessary transformations.

After fitting the preprocessor, we retrieve the transformed column names, which incorporate information about the transformed features. Finally, we apply the preprocessor to the dataset, resulting in a processed DataFrame.

Now with data prepared let’s head into predicting house prices and house categories based on input features, we implements a multi-task model for regression and classification tasks using PyTorch and PyTorch Lightning.

First, we define a custom dataset class that converts the processed features dataset and the regression and classification targets into PyTorch tensors. This custom dataset class implements the required methods for a dataset, such as \_\_len\_\_ and \_\_getitem\_\_, which enable easy access to the data during model training.

Next, we introduce the shared bottom model, which is a fully connected neural network with one hidden layer. This model takes the input dimension, hidden dimension, and activation function as parameters. During the forward pass, the shared bottom model applies linear transformation and the specified activation function to the input.

To combine the shared bottom model with separate output layers for regression and classification tasks, we create a multi-task model derived from PyTorch Lightning's LightningModule. The model's forward method passes the input through the shared bottom model, producing regression and classification outputs. During the training step, the model calculates the loss for both tasks: the root mean squared error (RMSE) for regression and the cross-entropy loss for classification. These losses are logged, and the total loss is returned. The validation step follows a similar process for the validation data.

For optimization, we utilize the Adam optimizer with a learning rate of 0.1. This choice of optimizer allows for efficient gradient-based optimization of the model's parameters.

Now, let's move on to the training process of the multi-task model. Firstly, we split the dataset into training and validation sets using the train\_test\_split function from the scikit-learn library. To work effectively with the training and validation sets, we create custom datasets using the CustomDataset class defined earlier. The X\_train, y\_regression\_train, and y\_classification\_train are passed to create the train\_dataset, while X\_val, y\_regression\_val, and y\_classification\_val are used to create the val\_dataset.

To facilitate efficient data loading during training, we instantiate data loaders using the DataLoader class from PyTorch. The train\_loader is created with the train\_dataset and a batch size of 32, and shuffle=True is set to shuffle the data during training. Similarly, the val\_loader is created with the val\_dataset and the same batch size.

To determine the number of classes for the classification task, we calculate the length of the unique values in the classification\_targets. This information is passed as the num\_classes parameter when creating an instance of the MyMultiTaskModel class. The input dimension is set to the number of features in the features dataset, hidden\_dim is set to 64, and both activation\_func\_regression and activation\_func\_classification are set to the ReLU activation function.

We create a PyTorch Lightning Trainer with a maximum number of epochs set to 10. The trainer orchestrates the training process, providing functionalities for monitoring, logging, and potential callbacks for improved training control.

Now, we train the model using the trainer.fit() method, which takes the model, train\_loader, and val\_loader as arguments. During training, the model iterates over the training data in batches, computes the loss, and updates the model's parameters. The validation data is used to evaluate the model's performance and monitor the loss metrics. The reported validation loss is 29,130.3184. Lower validation loss values typically indicate better model performance, as they signify a smaller discrepancy between the predicted and actual values.

However, we can further enhance the model's performance by conducting hyperparameter optimization. To achieve this, we employ Optuna, a Python library for automatic hyperparameter tuning. With Optuna, we create an Optuna study object using optuna.create\_study() and set the direction to "minimize" since we want to minimize the objective function. The study will search for the best hyperparameters through a specified number of trials.

We call the study.optimize() method, passing the objective function to be optimized and the number of trials as arguments. This initiates the optimization process, where Optuna explores different hyperparameter configurations and evaluates their performance.

After the optimization process is complete, we retrieve the best hyperparameters using study.best\_params, which returns a dictionary containing the hyperparameters that yielded the best results.

Next, we iterate over the activation functions and optimizers using nested loops. For each combination, we instantiate the optimizer class and set the activation function in the shared bottom model of our multi-task model.

Inside the loop, we create a new instance of the multi-task model using the best hyperparameters obtained from Optuna. We specify the best activation function for both the regression and classification tasks. Additionally, we create an optimizer with the best learning rate.

Then, we instantiate a new PyTorch Lightning Trainer, and we train the model using the trainer.fit() method, passing the updated model, training data loader, and validation data loader as arguments. The model is trained for the specified number of epochs, and the training progress is logged.

After the loop, we compare the validation losses for each combination of activation function and optimizer. If the current validation loss is better than the previously recorded best validation loss, we update the best validation loss, activation function, and optimizer accordingly.

Finally, we create a new instance of the multi-task model using the best hyperparameters. We specify the best activation function for both the regression and classification tasks. Additionally, we create an optimizer with the best learning rate.

We instantiate a new PyTorch Lightning Trainer, and the model is trained using the trainer.fit() method, passing the updated model, training data loader, and validation data loader as arguments. The model is trained for the specified number of epochs, and the training progress is logged.

After the training phase, we extract the validation loss from the logged metrics available within the trainer object. The validation loss serves as an evaluative metric, quantifying the model's performance on unseen data.

By applying Optuna's optimization algorithms, we systematically explored the hyperparameter space, leading to the identification of optimal hyperparameter configurations that minimize the validation loss. This iterative process ensured enhanced performance and improved generalization capabilities of the multi-task model when deployed on previously unseen data.

After tuning the hyperparameters, the best validation loss was significantly reduced to 27,436.1602, showcasing the effectiveness of hyperparameter optimization in improving the model's performance. This reduction in the validation loss demonstrates better alignment between the predicted and actual values, resulting in more accurate predictions of house prices and house categories.

In conclusion, the implementation of a multi-task model using PyTorch and PyTorch Lightning, coupled with hyperparameter optimization using Optuna, allowed us to build a robust model for predicting house prices and house categories. The optimized model achieved superior performance, outperforming the previous version by reducing the validation loss. This demonstrates the importance of hyperparameter tuning in maximizing the potential of deep learning models and improving their ability to generalize to unseen data.