GridSearch and Model Pipelines

Learning Objectives

- Use a model pipeline in GridSearchCV.

GridSearch and Model Pipelines

We will now demonstrate how to use GridSearchCV when using a model pipeline. This lesson will accomplish the same goals as the "GridSearchCV" lesson. In that lesson, we transformed the data and then "manually" fed that data into our model. Then we used, GridSearchCV on the model. In this lesson, we will combine the transformation and model into a model pipeline and use GridSearchCV on the model pipeline. Ultimately, you can use either approach, but you wouldn't use both (as they accomplish the same thing.) We tend to prefer model pipelines because they are more streamlined and efficient. You will also see that using a model pipeline provides the opportunity to tune our preprocessing parameters along with our model parameters.

Imports

```
# Imports
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
# Set pandas as the default output for sklearn
from sklearn import set_config
```

```
set_config(transform_output='pandas')
```

Define Custom Functions

```
def regression_metrics(y_true, y_pred, label='', verbose = True, output_dict=False):
  # Get metrics
  mae = mean_absolute_error(y_true, y_pred)
  mse = mean_squared_error(y_true, y_pred)
  rmse = mean_squared_error(y_true, y_pred, squared=False)
  r_squared = r2_score(y_true, y_pred)
  if verbose == True:
    # Print Result with Label and Header
    header = "-"*60
    print(header, f"Regression Metrics: {label}", header, sep='\n')
    print(f"- MAE = {mae:,.3f}")
    print(f"- MSE = {mse:,.3f}")
    print(f"- RMSE = {rmse:,.3f}")
    print(f"- R^2 = {r_squared:,.3f}")
  if output_dict == True:
      metrics = {'Label':label, 'MAE':mae,
                 'MSE':mse, 'RMSE':rmse, 'R^2':r_squared}
      return metrics

def evaluate_regression(reg, X_train, y_train, X_test, y_test, verbose = True,
                        output_frame=False):
  # Get predictions for training data
  y_train_pred = reg.predict(X_train)

  # Call the helper function to obtain regression metrics for training data
  results_train = regression_metrics(y_train, y_train_pred, verbose = verbose,
                                     output_dict=output_frame,
                                     label='Training Data')
  print()
  # Get predictions for test data
  y_test_pred = reg.predict(X_test)
  # Call the helper function to obtain regression metrics for test data
  results_test = regression_metrics(y_test, y_test_pred, verbose = verbose,
                                    output_dict=output_frame,
                                      label='Test Data' )

  # Store results in a dataframe if ouput_frame is True
  if output_frame:
    results_df = pd.DataFrame([results_train,results_test])
    # Set the label as the index
    results_df = results_df.set_index('Label')
```

```
    # Set index.name to none to get a cleaner looking result
    results_df.index.name=None
    # Return the dataframe
    return results_df.round(3)
```

Load Data

```
# Mount google drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
# Load Ames data from Dojo file structure
fpath = "/content/drive/MyDrive/CodingDojo/02-MachineLearning/Week05/Data/ames-housing-dojo-for-ml.csv"
df = pd.read_csv(fpath)
df = df.set_index("PID")
df.head()
```

PID	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Utilities	Neighborhood	Bldg Type	House Style	Overall Qual	...	Garage Area	Garage Qual	Garage Cond	Paved Drive	Fence	SalePrice	Month	Year	Total Full Baths	Total Half Baths
907227090	RL	60.0	7200	Pave	NaN	AllPub	CollgCr	1Fam	1Story	5	...	297.0	TA	TA	Y	MnPrv	119900.0	3	2006	1.0	0.0
527108010	RL	134.0	19378	Pave	NaN	AllPub	Gilbert	1Fam	2Story	7	...	576.0	TA	TA	Y	NaN	320000.0	3	2006	3.0	1.0
534275170	RL	NaN	12772	Pave	NaN	AllPub	NAmes	1Fam	1Story	6	...	301.0	TA	TA	Y	NaN	151500.0	4	2007	1.0	0.0
528104050	RL	114.0	14803	Pave	NaN	AllPub	NridgHt	1Fam	1Story	10	...	1220.0	TA	TA	Y	NaN	385000.0	6	2008	3.0	0.0
533206070	FV	32.0	3784	Pave	Pave	AllPub	Somerst	TwnhsE	1Story	8	...	476.0	TA	TA	Y	NaN	193800.0	2	2007	3.0	0.0

5 rows × 35 columns

Using a Model Pipeline

The next steps include the same steps needed to make and evaluate a model with a model pipeline. We have demonstrated this process in the "Model Pipelines" lesson with a different dataset. Here we go through the process with our Ames data.

1) Define X and y and Perform Validation Split

2) Define preprocessing pipelines for each group of features

3) Combine the preprocessing pipelines into the column transformed

4) Create the model pipeline using the column transformer and the model

5) Fit and evaluate the model

After we have gone through these steps, we will add GridSearchCV to tune the model.

Define X and y and Perform Validation Split

```
# Make a list of features to drop
drop_from_model = ['Utilities', # Quasi-constant
                   "Street", # Quasi-constant
                   'MS Zoning', # Stakeholder can't change
                   'Lot Frontage',  # Stakeholder can't change
                   'Lot Area', # Stakeholder can't change
                   'Neighborhood',  # Stakeholder can't change
                   'Year Built'] # Stakeholder can't change
# Define features matrix
X = df.drop(columns = [*drop_from_model,'SalePrice'])
# Define target
y = df['SalePrice']
# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 42)
# Preview training data
X_train.head()
```

PID	Alley	Bldg Type	House Style	Overall Qual	Overall Cond	Year Remodeled	Exter Qual	Exter Cond	Bsmt Unf Sqft	Total Bsmnt Sqft	...	Garage Cars	Garage Area	Garage Qual	Garage Cond	Paved Drive	Fence	Month	Year	Total Full Baths	Total Half Baths
905475520	NaN	1Fam	1Story	4	5	1994	TA	TA	0.0	0.0	...	1.0	308.0	TA	TA	N	NaN	8	2007	1.0	0.0
909254010	NaN	1Fam	2Story	7	8	1990	TA	TA	600.0	600.0	...	1.0	215.0	Fa	TA	Y	MnPrv	5	2009	1.0	0.0
531450090	NaN	1Fam	1Story	6	5	1991	TA	TA	78.0	1278.0	...	2.0	496.0	TA	TA	Y	GdWo	6	2008	3.0	0.0
903400040	Pave	1Fam	2Story	6	6	1950	TA	TA	764.0	764.0	...	2.0	520.0	TA	TA	N	GdPrv	7	2007	1.0	0.0
527107130	NaN	1Fam	SLvl	7	5	1997	TA	TA	100.0	384.0	...	2.0	390.0	TA	TA	Y	NaN	6	2009	2.0	1.0

Create Pipelines for Transformation

Note that this is the same code we have been using for preprocessing the Ames data.

```
# PREPROCESSING PIPELINE FOR NUMERIC DATA
# Save list of column names
num_cols = X_train.select_dtypes("number").columns
print("Numeric Columns:", num_cols)
# instantiate preprocessors
impute_median = SimpleImputer(strategy='median')
scaler = StandardScaler()
# Make a numeric preprocessing pipeline
num_pipe = make_pipeline(impute_median, scaler)
# Making a numeric tuple for ColumnTransformer
```

```
num_tuple = ('numeric', num_pipe, num_cols)
```

```
Numeric Columns: Index(['Overall Qual', 'Overall Cond', 'Year Remodeled', 'Bsmt Unf Sqft',
       'Total Bsmnt Sqft', 'Living Area Sqft', 'Bedroom', 'Kitchen',
       'Total Rooms', 'Garage Yr Blt', 'Garage Cars', 'Garage Area', 'Month',
       'Year', 'Total Full Baths', 'Total Half Baths'],
      dtype='object')
```

Make the Ordinal pipeline.

```
# Save list of column names
ord_cols = ['Exter Qual','Exter Cond', 'Garage Qual',"Garage Cond"]
print("Ordinal Columns:", ord_cols)
# Create imputer for ordinal data
impute_na_ord = SimpleImputer(strategy='constant', fill_value='NA')
# Making the OrdinalEncoder
# Specifying order of categories for our  Ordinal Qual/Cond Columms
qual_cond_order = ['NA','Po', 'Fa', 'TA', 'Gd', 'Ex']
# Making the list of order lists for OrdinalEncoder
ordinal_category_orders = [qual_cond_order, qual_cond_order,
                           qual_cond_order, qual_cond_order]
ord_encoder = OrdinalEncoder(categories=ordinal_category_orders)
# Making a final scaler to scale category #'s
scaler_ord = StandardScaler()
# Making an ord_pipe
ord_pipe = make_pipeline(impute_na_ord, ord_encoder, scaler_ord)
# Making an ordinal_tuple for ColumnTransformer
ord_tuple = ('ordinal', ord_pipe, ord_cols)
```

```
Ordinal Columns: ['Exter Qual', 'Exter Cond', 'Garage Qual', 'Garage Cond']
```

```
# PREPROCESSING PIPELINE FOR ONE-HOT-ENCODED DATA
# Save list of column names
ohe_cols = X_train.select_dtypes('object').drop(columns=ord_cols).columns
print("OneHotEncoder Columns:", ohe_cols)
# Instantiate the individual preprocessors
impute_na = SimpleImputer(strategy='constant', fill_value = "NA")
ohe_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
# Make pipeline with imputer and encoder
ohe_pipe = make_pipeline(impute_na, ohe_encoder)
# Making a ohe_tuple for ColumnTransformer
ohe_tuple = ('categorical', ohe_pipe, ohe_cols)
```

Instantiate the Column Transformer

```
# Create the Column Transformer
preprocessor = ColumnTransformer([num_tuple, ord_tuple, ohe_tuple],
                                 verbose_feature_names_out=False)
```

Remember, we won't fit the transformer yet. We will fit the preprocessor and the model in one step once it is in the pipeline.

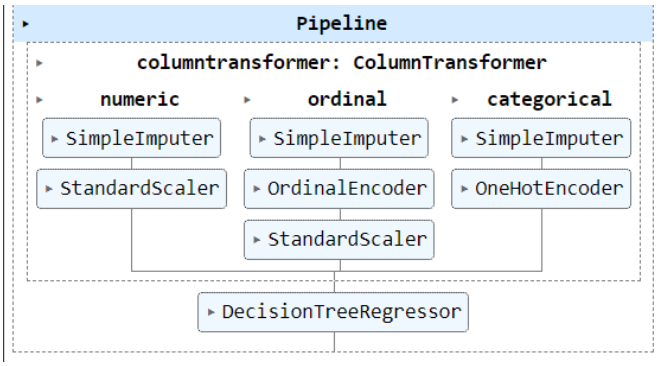
Instantiate Model

```
# We will start with a default model
# For reproducible results, set the random state
dec_tree = DecisionTreeRegressor(random_state = 42)
```

Create Model Pipeline

```
# Combine the preprocessor with the decision tree model in a model pipeline
dec_tree_pipe = make_pipeline(preprocessor, dec_tree)
```

```
# Fit the model on the training data only
dec_tree_pipe.fit(X_train, y_train)
```




```
# Make predictions and evalute with custom function
evaluate_regression(dec_tree_pipe, X_train, y_train, X_test, y_test)
```

```
-------------------------------------------------------------
Regression Metrics: Training Data
-------------------------------------------------------------
- MAE = 3.778
```

```
- MSE = 15,678.198
- RMSE = 125.213
- R^2 = 1.000
------------------------------------------------------------
Regression Metrics: Test Data
------------------------------------------------------------
- MAE = 24,817.956
- MSE = 1,631,506,619.075
- RMSE = 40,391.913
- R^2 = 0.667
```

GridSearchCV with Model Pipelines

Select Parameters for Tuning and Define the Param Grid

The first difference when using a model pipeline occurs when you investigate the parameters available for tuning.

```
# Looking at options for tuning this model
dec_tree_pipe.get_params()
```

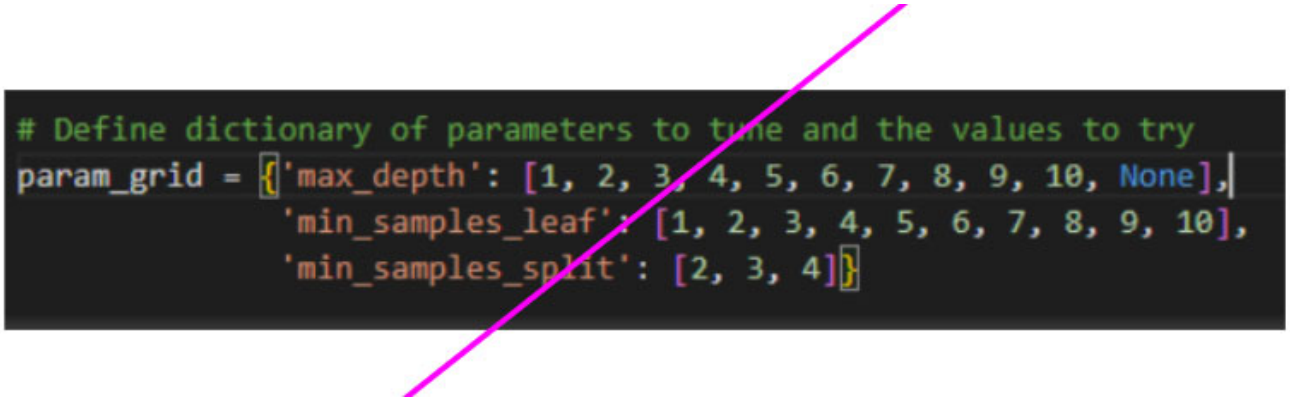
```
{'memory': None,
 'steps': [('columntransformer',
   ColumnTransformer(transformers=[('numeric',
                                    Pipeline(steps=[('simpleimputer',
                                                     SimpleImputer(strategy='median')),
                                                    ('standardscaler',
                                                     StandardScaler())]),
 ---Output removed to reduce space---
 'columntransformer__categorical__onehotencoder__sparse': 'deprecated',
 'columntransformer__categorical__onehotencoder__sparse_output': False,
 'decisiontreeregressor__ccp_alpha': 0.0,
 'decisiontreeregressor__criterion': 'squared_error',
 'decisiontreeregressor__max_depth': None,
 'decisiontreeregressor__max_features': None,
 'decisiontreeregressor__max_leaf_nodes': None,
 'decisiontreeregressor__min_impurity_decrease': 0.0,
 'decisiontreeregressor__min_samples_leaf': 1,
 'decisiontreeregressor__min_samples_split': 2,
 'decisiontreeregressor__min_weight_fraction_leaf': 0.0,
 'decisiontreeregressor__random_state': 42,
 'decisiontreeregressor__splitter': 'best'}
```

First, the output is actually much longer, but we have cut out the middle to conserve space. We now see parameters for each step in the process. For example, in the abbreviate output, we see the parameters for decisiontreeregressor at the end, but just above those, we also see column_transfomer__categorical__onehotencoder parameters. We can try different parameters for both our model and preprocessing steps in GridSearchCV.

Notice that the format is: step__hyperparameter. A double underscore, also called a ‘dunder,’ separates the step name and the hyperparameter. All of the parameters available for our model start with

"decisiontreeregressor__." The parameters themselves are the same as we have seen (i.e., max_depth, min_samples_leaf, etc.).

When we defined our param grid previously (when not using a model pipeline), it looked like this:



When using a model pipeline, we must specify the entire name in our param grid to indicate the step along with the parameter.

```
# Define dictionary of parameters to tune and the values to try
param_grid = {'decisiontreeregressor__max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, None],
              'decisiontreeregressor__min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
              'decisiontreeregressor__min_samples_split': [2, 3, 4]}
```

Instantiate GridSearchCV

Now we include the model pipeline (instead of just the model) as the first argument when we instantiate our GridSearchCV.

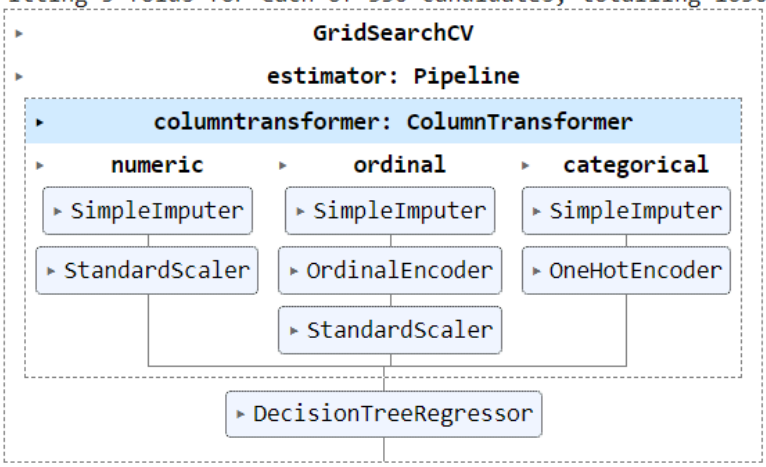
```
# Instantiate GridSearchCV
grid_search = GridSearchCV(dec_tree_pipe, param_grid, n_jobs = -1, verbose = 1)
```

The remaining steps are the same whether you use a pipeline or not.

Fit the GridSearch on the Training Data

```
# Fit the Gridsearch on the training data
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 330 candidates, totalling 1650 fits
```




```
# Obtain the best combination directly
grid_search.best_params_
```

```
{'decisiontreeregressor__max_depth': 10,
 'decisiontreeregressor__min_samples_leaf': 8,
 'decisiontreeregressor__min_samples_split': 2}
```

Define the Best Model

Recall that this will automatically refit the model on the entire training set (no folds.)

```
# Now define the best version of the model
best_model = grid_search.best_estimator_
```

Predict and Evaluate

```
# Predict and Evaluate with custom function
evaluate_regression(best_model, X_train, y_train, X_test, y_test)
```

```
------------------------------------------------------------
Regression Metrics: Training Data
------------------------------------------------------------
- MAE = 15,556.335
- MSE = 627,110,888.699
- RMSE = 25,042.182
- R^2 = 0.909
------------------------------------------------------------
Regression Metrics: Test Data
------------------------------------------------------------
- MAE = 21,185.922
- MSE = 999,117,593.978
- RMSE = 31,608.821
- R^2 = 0.796
```

Notice the results are the same as in the previous lesson. The only difference was that we used a model pipeline.

Summary

This lesson demonstrated how to use a model pipeline with GridSearchCV. After creating the model pipeline, the biggest difference occurs when creating the param grid. Rather than just defining the parameter (such as" max_depth"), we must also provide the step of the parameter (such as "decisiontreeregressor_max_depth"). While we only tuned the parameters of the model itself in this lesson, we could also tune the parameters associated with the processing steps if we wished. We also must include the model pipeline as the first argument in the GridSearchCV instead of just the model.