Machine Learning

|Select and Train a Model

Prof. Dr. Mostafa Elhosseini

https://youtube.com/drmelhosseini

Agenda

- Select and Train a Model
- Linear Regression
- Decision Tree Regressor
- Train and Evaluate on the Training Set
- Better Evaluation Using Cross-Validation
- Random Forest Regressor
- Ensembles Models

Transformation Pipelines

- You framed the problem
  - You got the data and explored it,
  - You sampled a training set and a test set, and
  - You wrote a preprocessing pipeline to automatically clean up and prepare your data for machine learning algorithms.
- You are now ready to select and train a machine learning model

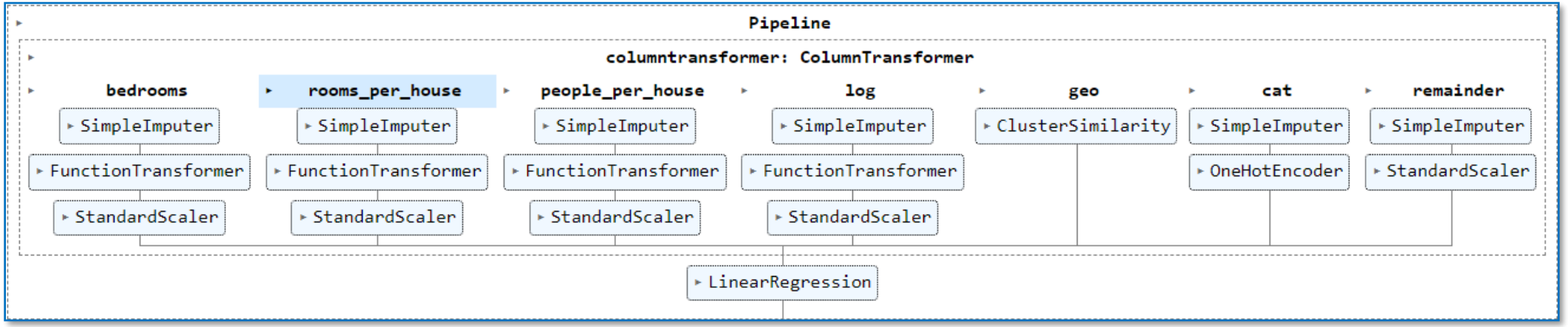
Recall

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"]  # feature names out

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler())

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                     StandardScaler())
preprocessing = ColumnTransformer([
        ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
        ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
        ("people_per_house", ratio_pipeline(), ["population", "households"]),
        ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                               "households", "median_income"]),
        ("geo", cluster_simil, ["latitude", "longitude"]),
        ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
    ],
    remainder=default_num_pipeline)  # one column remaining: housing_median_age
```

Train and Evaluate on the Training Set

- A very basic Linear regression model

```
from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)
```



Train and Evaluate on the Training Set

- Try the full preprocessing pipeline on a few training instances:

```
housing_predictions = lin_reg.predict(housing)
housing_predictions[:5].round(-2)  # -2 = rounded to the nearest hundred
```

```
array([243700., 372400., 128800.,  94400., 328300.])
```

- Compare against the actual values:

```
housing_labels.iloc[:5].values
```

```
array([458300., 483800., 101700.,  96100., 361800.])
```

Train and Evaluate on the Training Set

- Compare against the actual values:

```
array([243700., 372400., 128800.,  94400., 328300.])
```

```
array([458300., 483800., 101700.,  96100., 361800.])
```

```
error_ratios = housing_predictions[:5].round(-2) / housing_labels.iloc[:5].values - 1
print(", ".join([f"{100 * ratio:.1f}%" for ratio in error_ratios]))
```

```
-46.8%, -23.0%, 26.6%, -1.8%, -9.3%
```

Train and Evaluate on the Training Set

- RMSE on the whole training set using Scikit-Learn's

```
from sklearn.metrics import mean_squared_error

lin_rmse = mean_squared_error(housing_labels, housing_predictions,
                              squared=False)
lin_rmse
```

```
68687.89176590038
```

- This is better than nothing, but clearly not a great score
- This is an example of a model underfitting the training data

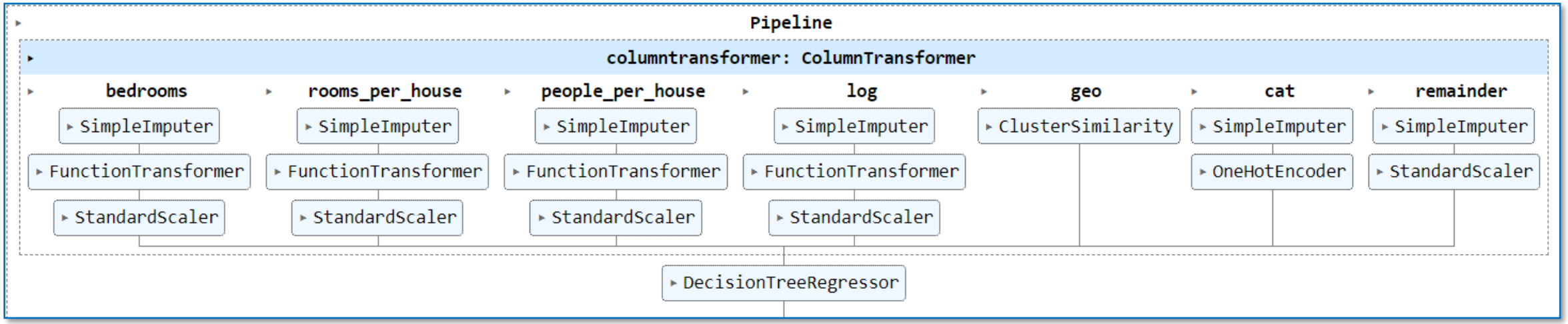
Train and Evaluate on the Training Set

- This is an example of a model underfitting the training data
- When this happens, it can mean that:
  - The features do not provide enough information to make good predictions, or
  - The model is not powerful enough.
- As we saw in the previous chapter, the main ways to fix underfitting are to **select a more powerful model**, to feed the training algorithm with **better features**, or to **reduce the constraints** on the model
  - Revise Lecture 14 – Chapter 01

Train and Evaluate on the Training Set

- You decide to try a DecisionTreeRegressor

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)
```



Train and Evaluate on the Training Set

- Now that the model is trained, you evaluate it on the training set

```
housing_predictions = tree_reg.predict(housing)
tree_rmse = mean_squared_error(housing_labels, housing_predictions,
                               squared=False)
tree_rmse
```

```
0.0
```

- Could this model really be absolutely perfect?
- It is much more likely that the **model has badly overfit** the data. How can you be sure?

Train and Evaluate on the Training Set

- you **don’t want to touch the test set** until you are ready to launch a model you are confident about, so you need to use **part of** the training set for **training** and part of it for **model validation**

Better Evaluation Using Cross-Validation

- use the **train_test_split()** function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set
- A great alternative is to use Scikit-Learn's k_-fold cross-validation feature.
- The following code randomly splits the training set into 10 nonoverlapping subsets called folds, then it trains and evaluates the decision tree model 10 times, picking a different fold for evaluation every time and using the other 9 folds for training. The result is an array containing the 10 evaluation scores

Better Evaluation Using Cross-Validation

```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring="neg_root_mean_squared_error", cv=10)
```

- Scikit-Learn's cross-validation features expect a **utility function** (greater is better) rather than a cost function (lower is better), so the **scoring function is actually the opposite** of the RMSE

```
pd.Series(tree_rmses).describe()
```

```
count       10.000000
mean     66868.027288
std       2060.966425
min      63649.536493
25%      65338.078316
50%      66801.953094
75%      68229.934454
max      70094.778246
dtype: float64
```

Better Evaluation Using Cross-Validation

```
lin_rmses = -cross_val_score(lin_reg, housing, housing_labels,
                             scoring="neg_root_mean_squared_error", cv=10)
pd.Series(lin_rmses).describe()
```

```
count       10.000000
mean     69858.018195
std       4182.205077
min      65397.780144
25%      68070.536263
50%      68619.737842
75%      69810.076342
max      80959.348171
dtype: float64
```

Better Evaluation Using Cross-Validation

- Now the decision tree doesn't look as good as it did earlier. In fact, it seems to perform almost as poorly as the linear regression model!
- Notice that crossvalidation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its **standard deviation**).

Better Evaluation Using Cross-Validation

- The decision tree has an RMSE of about 66,868, with a standard deviation of about 2,061. You would not have this information if you just used one validation set. But cross-validation comes at the cost of training the model several times, so it is not always feasible.
- There’s an overfitting problem because the training error is low (actually zero) while the validation error is high.

Random Forest Regressor

- Random Forest is a method used in machine learning that involves creating a "forest" of decision trees to make better predictions. Imagine you're trying to make a tough decision, like choosing the best route to take on a road trip.
- Instead of relying on one friend's advice, you ask a bunch of friends, each with a slightly different perspective. Each friend acts like a "tree" in this forest, offering their route based on their own experiences and knowledge.

Random Forest Regressor

- In machine learning, a decision tree is a model that makes decisions by asking a series of yes-or-no questions about the data (like "Is it raining?" or "Is it rush hour?") to reach a conclusion (the best route). However, a single tree can sometimes make mistakes or focus too much on the specific details of the data it was trained on, leading to poor decisions when it encounters new situations.

Random Forest Regressor

- Random Forest improves on this by creating many decision trees, each trained on a random subset of the data and features (or questions it can ask). When it's time to make a prediction, the Random Forest takes the majority vote from all its trees, making the final decision more robust and accurate than any single tree's decision. This process is like getting a consensus from all your friends, which often leads to a better decision for your road trip.

Random Forest Regressor

- This technique is powerful for both classification (categorizing things) and regression (predicting a continuous value, like house prices) tasks in machine learning, providing high accuracy while handling a large amount of data with diverse variables.

Random Forest Regressor

- In machine learning, **ensembles** are methods that combine the predictions from multiple models to improve accuracy and robustness over individual models. Think of it as a team effort in decision-making, where instead of relying on a single person's judgment, you combine insights from several people to reach a better conclusion. This approach often leads to more reliable and accurate predictions because it reduces the likelihood of making mistakes that a single model might make.

Random Forest Regressor

- Ensembles work on the principle that a group of models, each with its own strengths and weaknesses, can complement each other. When their predictions are combined, the ensemble can often correct the errors of individual models, leading to better overall performance. There are a few key types of ensemble methods:
  - Bagging (Bootstrap Aggregating)
  - Boosting
  - Stacking (Stacked Generalization)

Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                                scoring="neg_root_mean_squared_error", cv=10)
```

```
pd.Series(forest_rmses).describe()
```

```
count       10.000000
mean     47019.561281
std       1033.957120
min      45458.112527
25%      46464.031184
50%      46967.596354
75%      47325.694987
max      49243.765795
dtype: float64
```

Random Forest Regressor

- Let's compare this RMSE measured using cross-validation (the "validation error") with the RMSE measured on the training set (the "training error"):

```
forest_reg.fit(housing, housing_labels)
housing_predictions = forest_reg.predict(housing)
forest_rmse = mean_squared_error(housing_labels, housing_predictions,
                                 squared=False)
forest_rmse
```

```
17474.619286483998
```

- The training error is much lower than the validation error, which usually means that the model has overfit the training set. Another possible explanation may be that there's a mismatch between the training data and the validation data, but it's not the case here, since both came from the same dataset that we shuffled and split in two parts.

Random Forest Regressor

- Possible solutions are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. Before you dive much deeper into random forests, however, you should try out many other models from various categories of machine learning algorithms (e.g., several support vector machines with different kernels, and possibly a neural network), without spending too much time tweaking the hyperparameters.
- The goal is to shortlist a few (two to five) promising models.

What is Next?

- Fine-Tune Your Model