Select and Train a Model

At last! 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.

Training and Evaluating on the Training Set

The good news is that thanks to all these previous steps, things are now going to be easy! You decide to train a very basic Linear Regression model to get started:

```
from sklearn.linear_model import LinearRegression

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

Done! You now have a working Linear Regression model. Let's try it out on the training set, looking at the first 5 predictions and comparing them to the labels:

```
>>> housing_predictions = lin_reg.predict(housing)
>>> housing_predictions[:5].round(-2)  # -2 = rounded to the nearest
hundred
array([243700., 372400., 128800.,  94400., 328300.])
>>> housing_labels.iloc[:5].values
array([458300., 483800., 101700.,  96100., 361800.])
```

Well, it works, but not always: the first prediction is way off (by over $200,000!), while the other predictions are better: two are off by about 25%, and two are off by less than 10%. Remember that you chose to use the RMSE as your performance measure, so you want to measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error()` function, with the `squared` argument set to `False`:

```
>>> from sklearn.metrics import mean_squared_error
>>> lin_rmse = mean_squared_error(housing_labels,
housing_predictions,
...                               squared=False)
...
>>> lin_rmse
68687.89176589991
```

This is better than nothing, but clearly not a great score: most districts' `median_housing_values` range between $120,000 and $265,000, so a typical prediction error of $68,628 is really not very satisfying. 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 that 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. This model is not regularized, which rules out the last option. You could try to add more features, but first you want to try a more complex model to see how it does.

So you decide to try a `DecisionTreeRegressor`, as this is a fairly powerful model capable of finding complex nonlinear relationships in the data (Decision Trees are presented in more detail in Chapter 6):

```
from sklearn.tree import DecisionTreeRegressor

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

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
```

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data. How

can you be sure? As we saw earlier, 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

One way to evaluate the Decision Tree model would be to 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. It's a bit of work, but nothing too difficult, and it would work fairly well.

A great alternative is to use Scikit-Learn's *K-fold cross-validation* feature. The following code randomly splits the training set into 10 non-overlapping 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:

```
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)
```

> **WARNING**
>
> 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. It's a negative value, which is why we need to switch the sign of the output to get the RMSE scores.

Let's look at the results:

```
>>> 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
```

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 cross-validation 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). 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.

If you compute the same metric for the Linear Regression model, you will find that the mean RMSE is 69,858 and the standard deviation is 4,182. So the Decision Tree model seems to perform very slightly better than the linear model, but only marginally better due to severe overfitting. We know there's an overfitting problem because the training error is low (actually zero) while the validation error is high.

Let's try one last model now: the `RandomForestRegressor`. As we will see in Chapter 7, Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Such models composed of many other models are called *ensembles*: they are capable of boosting the performance of the underlying model (in this case, Decision Trees). The code is much the same as earlier:

```
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)
```

Let's look at these scores:

```
>>> 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
```

Wow, this is much better: Random Forests really look very promising for this task! However, if you train a `RandomForest` and measure the RMSE on the training set, you will find about 17,474: that's much lower, meaning that there's still quite a lot of overfitting going on. 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.

Fine-Tune Your Model

Let's assume that you now have a shortlist of promising models. You now need to fine-tune them. Let's look at a few ways you can do that.

Grid Search

One option would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations.

Instead, you can use Scikit-Learn's `GridSearchCV` class to search for you. All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code searches for the best combination of hyperparameter values for the `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
```

```
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid = [
    {'preprocessing__geo__n_clusters': [5, 8, 10],
     'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo__n_clusters': [10, 15],
     'random_forest__max_features': [6, 8, 10]},
]
grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```

Notice that you can refer to any hyperparameter of any estimator in a pipeline, even if this estimator is nested deep inside several pipelines and column transformers. For example, when Scikit-Learn sees `"preprocessing__geo__n_clusters"`, it splits this string at the double underscores, then it looks for an estimator named `"preprocessing"` inside the pipeline and finds the preprocessing `ColumnTransformer`. Then it looks for a transformer named `"geo"` inside this `ColumnTransformer` and finds the `ClusterSimilarity` transformer we used on the latitude and longitude attributes. The it finds this transformer's `n_clusters` hyperparameter. Similarly, `random_forest__max_features` refers to the `max_features` hyperparameter of the estimator named "random_forest", which is of course the `RandomForest` model (the `max_features` hyperparameter will be explained in Chapter 7).

> **TIP**
>
> Wrapping preprocessing steps in a Scikit-Learn pipeline allows you to tune the preprocessing hyperparameters along with the model hyperparameters. This is a good thing since they often interact. For example, perhaps increasing `n_clusters` requires increasing `max_features` as well. If fitting the pipeline transformers is computationally expensive, you can set the pipeline's `memory` hyperparameter to the path of a caching directory: when you first fit the pipeline, Scikit-Learn will save the fitted transformers to this directory. If you then fit the pipeline again with the same hyperparameters, Scikit-Learn will just load the cached transformers.

There are two dictionaries in this `param_grid`, so `GridSearchCV` will first evaluate all 3 × 3 = 9 combinations of `n_clusters` and `max_features`

hyperparameter values specified in the first `dict`, then it will try all 2 × 3 = 6 combinations of hyperparameter values in the second `dict`. So in total the grid search will explore 9 + 6 = 15 combinations of hyperparameter values, and it will train the pipeline 3 times per combination, since we are using three-fold cross validation. So there will be a grand total of 15 × 3 = 45 rounds of training! It may take a while, but when it is done you can get the best combination of parameters like this:

```
>>> grid_search.best_params_
{'preprocessing__geo__n_clusters': 15, 'random_forest__max_features':
6}
```

In this example, the best model is obtained by setting `n_clusters` to 15 and `max_features` to 8.

> **TIP**
>
> Since 15 is the maximum value that was evaluated for `n_clusters`, you should probably try searching again with higher values; the score may continue to improve.

The best estimator is available using `grid_search.best_estimator_`. If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea, since feeding it more data will likely improve its performance.

The evaluation scores are available using `grid_search.cv_results_`. This is a dictionary, but if you wrap it in a DataFrame, you get a nice list of all the test scores for each combination of hyperparameters and for each cross-validation split, as well as the mean test score across all splits.

```
>>> cv_res = pd.DataFrame(grid_search.cv_results_)
>>> cv_res.sort_values(by="mean_test_score", ascending=False,
inplace=True)
>>> [...]  # change column names to fit on this page, and show rmse =
-score
>>> cv_res.head()
   n_clusters max_features  split0  split1  split2  mean_test_rmse
12         15            6   43460   43919   44748           44042
```

```
13          15            8   44132   44075   45010           44406
14          15           10   44374   44286   45316           44659
7           10            6   44683   44655   45657           44999
9           10            6   44683   44655   45657           44999
```

The mean test RMSE score for the best model is 44,042, which is better than the score you got earlier using the default hyperparameter values (which was 47,019). Congratulations, you have successfully fine-tuned your best model!

Randomized Search

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but `RandomizedSearchCV` is often preferable, especially when the hyperparameter search space is large. This class can be used in much the same way as the `GridSearchCV` class, but instead of trying out all possible combinations, it evaluates a fixed number of combinations, selecting a random value for each hyperparameter at every iteration. This may sound surprising, but this approach has several benefits:

- If some of your hyperparameters are continuous (or discrete but with many possible values), and you let randomized search run for, say, 1,000 iterations, then it will explore 1,000 different values for each of these hyperparameters, whereas grid search would only explore the few values you listed for each one.
- Suppose a hyperparameter does not actually make much difference, but you don’t know it yet. If it has 10 possible values you add it to your grid search, then training will take 10 times longer. But if you add it to a random search, it will not make any difference.
- Suppose there are 6 hyperparameters to explore, each with 10 possible values, then grid search offers no other choice than training the model a million times, whereas random search can always run for any number of iterations you choose.

For each hyperparameter, you must provide either a list of possible values, or a probability distribution:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {'preprocessing__geo__n_clusters': randint(low=3,
high=50),
                  'random_forest__max_features': randint(low=2,
high=20)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distribs, n_iter=10,
cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

Scikit-Learn also has two other hyperparameter search classes: `HalvingRandomSearchCV` and `HalvingGridSearchCV`. Their goal is to use the computational resources more efficiently, either to train faster or to explore a larger hyperparameter space. Here's how they work: in the first round, many hyperparameter combinations (called "candidates") are generated using either the grid approach or the random approach. These candidates are then used to train models which are then evaluated using cross-validation, as usual. However, training uses limited resources which speeds up this first round considerably. By default, "limited resources" means that the models are trained on a small part of the training set. However, other limitations are possible, such as reducing the number of training iterations if the model has a hyperparameter to set it. Once every candidate has been evaluated, only the best ones go on to the second round, where they are allowed more resources to compete. After several rounds, the final candidates are evaluated using full resources. This may save you some time tuning hyperparameters.

Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model—just like Random Forests perform better than the individual Decision Trees they rely on—especially if the individual models make very different types of errors. For example, you could train and fine-tune a k-Nearest Neighbors model, then create an ensemble model that just predicts the mean of

the random forest prediction and the KNN's prediction. We will cover this topic in more detail in Chapter 7.

Analyze the Best Models and Their Errors

You will often gain good insights on the problem by inspecting the best models. For example, the `RandomForestRegressor` can indicate the relative importance of each attribute for making accurate predictions:

```
>>> final_model = rnd_search.best_estimator_  # includes
preprocessing
>>> feature_importances =
final_model["random_forest"].feature_importances_
>>> feature_importances.round(2)
array([0.07, 0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, [...], 0.01])
```

Let's sort these importance scores in descending order and display them next to their corresponding attribute names:

```
>>> sorted(zip(feature_importances,
...            final_model["preprocessing"].get_feature_names_out()),
...            reverse=True)
...
[(0.18694559869103852, 'log__median_income'),
 (0.0748194905715524, 'cat__ocean_proximity_INLAND'),
 (0.06926417748515576, 'bedrooms_ratio__bedrooms_ratio'),
 (0.05446998753775219, 'rooms_per_house__rooms_per_house'),
 (0.05262301809680712, 'people_per_house__people_per_house'),
 (0.03819415873915732, 'geo__Cluster 0 similarity'),
 [...]
 (0.00015061247730531558, 'cat__ocean_proximity_NEAR BAY'),
 (7.301686597099842e-05, 'cat__ocean_proximity_ISLAND')]
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one `ocean_proximity` category is really useful, so you could try dropping the others).

> **TIP**
>
> The `sklearn.feature_selection.SelectFromModel` transformer can automatically drop the least useful features for you: when you fit it, it trains a model (typically a random forest), it looks at its `feature_importances_` attribute, and it selects the most useful features. Then when you call `transform()`, it just drops the other features.

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem: adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.

Now is also a good time to ensure that your model not only works well on average, but also on all categories of districts, whether they're rural or urban, rich or poor, North or South, minority or not, etc. This requires a bit of work creating subsets of your validation set for each category, but it's important: if your model performs poorly on a whole category of districts, then it should probably not be deployed until the issue is solved, or at least it should not be used to make predictions for that category, as it may do more harm than good.

Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. You are ready to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from your test set, and run your `final_model` to transform the data and make predictions, then evaluate these predictions:

```
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

final_predictions = final_model.predict(X_test)

final_rmse = mean_squared_error(y_test, final_predictions,
squared=False)
print(final_rmse)  # prints 41424.40026462184
```

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model

currently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% *confidence interval* for the generalization error using `scipy.stats.t.interval()`. We get a fairly large interval from 39,275 to 43,467, and our previous point estimate of 41,424 is roughly in the middle of it:

```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                          loc=squared_errors.mean(),
...                          scale=stats.sem(squared_errors)))
...
array([39275.40861216, 43467.27680583])
```

If you did a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation. That's because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets. It is not the case in this example since the test RMSE is lower than the validation RMSE, but when it happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

Now comes the project prelaunch phase: you need to present your solution (highlighting what you have learned, what worked and what did not, what assumptions were made, and what your system's limitations are), document everything, and create nice presentations with clear visualizations and easy-to-remember statements (e.g., "the median income is the number one predictor of housing prices"). In this California housing example, the final performance of the system is not much better than the experts' price estimates which were often off by 30%, but it may still be a good idea to launch it, especially if this frees up some time for the experts so they can work on more interesting and productive tasks.

Launch, Monitor, and Maintain Your System