5.
Select and Train a Model

Train a Linear Regression Model

- Train a Linear Regression model:

```python
from sklearn.linear_model import LinearRegression

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

- Try it out on a few instances from the training set and compare to labels:

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

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

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

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

Measure the Model's Error

- To measure the 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
```

- What is the problem?
- Underfitting! Try a better model.

Try Decision Tree Regressor

- Decision tree regressor is a powerful model, capable of finding complex nonlinear relationships in the data:

```
from sklearn.tree import DecisionTreeRegressor

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

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

```
0.0
```

- You will reach a 0.0 error! What happened?
  - The model has badly **overfit** the data.
- Divide the training set to *training* and *validation* parts.

Using Cross-Validation

- We can use the `train_test_split()` function to split the training set into a smaller training set and a validation set.
- Alternative: use Scikit-Learn's *k-fold cross-validation* feature.
  - Randomly split the training set into 10 distinct subsets called *folds*.
  - Train and evaluate the decision tree model 10 times.
    - Picking a different fold for evaluation every time and use the other 9 folds for training.

```python
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)
```

Using Cross-Validation

RMSE summary for 10-fold cross validation of decision tree model

```
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
```

RMSE summary for 10-fold cross validation of linear regression model

```
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
```

Try 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
```

```
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
```

6.
Fine-Tune Your Model

Grid Search

- You have a shortlist of promising models and you need to fine-tune them by fiddling their hyperparameters.

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

Grid Search

- Get the best parameters:

```
grid_search.best_params_
```

```
{'max_features': 8, 'n_estimators': 30}
```

- Get the best estimator:

```
grid_search.best_estimator_
```

```
RandomForestRegressor(max_features=8, n_estimators=30, random_state=42)
```

- Since 8 and 30 are the maximum values that were evaluated, you should probably try searching again with higher values.

Grid Search

```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
63669.11631261028 {'max_features': 2, 'n_estimators': 3}
55627.099719926795 {'max_features': 2, 'n_estimators': 10}
53384.57275149205 {'max_features': 2, 'n_estimators': 30}
60965.950449450494 {'max_features': 4, 'n_estimators': 3}
52741.04704299915 {'max_features': 4, 'n_estimators': 10}
50377.40461678399 {'max_features': 4, 'n_estimators': 30}
58663.93866579625 {'max_features': 6, 'n_estimators': 3}
52006.19873526564 {'max_features': 6, 'n_estimators': 10}
50146.51167415009 {'max_features': 6, 'n_estimators': 30}
57869.25276169646 {'max_features': 8, 'n_estimators': 3}
51711.127883959234 {'max_features': 8, 'n_estimators': 10}
49682.273345071546 {'max_features': 8, 'n_estimators': 30}
62895.06951262424 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.176157539405 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.40652318466 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52724.9822587892 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.5691951261 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.495668875716 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Grid Search – more details

```
pd.DataFrame(grid_search.cv_results_)
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_features	param_n_estimators	param_bootstrap	params	split0_test_sc
0	0.050905	0.004097	0.002766	0.000256	2	3	NaN	{'max_features': 2, 'n_estimators': 3}	-3.837622e
1	0.143706	0.002170	0.007205	0.000304	2	10	NaN	{'max_features': 2, 'n_estimators': 10}	-3.047771e
2	0.410306	0.004403	0.019903	0.000964	2	30	NaN	{'max_features': 2, 'n_estimators': 30}	-2.689185e
⋮									
17	0.370459	0.017424	0.007863	0.000056	4	10	False	{'bootstrap': False, 'max_features': 4, 'n_est...	-2.525578e

18 rows × 23 columns

Randomized Search

- When the hyperparameter search space is large, we prefer `RandomizedSearchCV` to `GridSearchCV`.
- Instead of all possible combinations, randomized search evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration.

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

param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distribs,
                                n_iter=10, cv=5, scoring='neg_mean_squared_error', random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

Randomized Search

```
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
49150.70756927707 {'max_features': 7, 'n_estimators': 180}
51389.889203389284 {'max_features': 5, 'n_estimators': 15}
50796.155224308866 {'max_features': 3, 'n_estimators': 72}
50835.13360315349 {'max_features': 5, 'n_estimators': 21}
49280.9449827171 {'max_features': 7, 'n_estimators': 122}
50774.90662363929 {'max_features': 3, 'n_estimators': 75}
50682.78888164288 {'max_features': 3, 'n_estimators': 88}
49608.99608105296 {'max_features': 5, 'n_estimators': 100}
50473.61930350219 {'max_features': 3, 'n_estimators': 150}
64429.84143294435 {'max_features': 5, 'n_estimators': 2}
```

Feature Importance

- *Feature importance* refers to techniques that assign a score to input features based on how useful they are at predicting a target variable.
  - Feature importance scores provide insight into the dataset and the model.
  - Feature importance can be used to improve a predictive model.

- The `RandomForestRegressor` can indicate the relative importance of each attribute for making accurate predictions:

```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances.round(2)
```

```
array([0.07, 0.06, 0.04, 0.02, 0.02, 0.02, 0.01, 0.38, 0.05, 0.11, 0.05,
       0.01, 0.17, 0.  , 0.  , 0.  ])
```

Feature Importance

```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

```
[(0.36615898061813423, 'median_income'),
 (0.1647809935615905, 'INLAND'),
 (0.10879295677551575, 'pop_per_hhold'),
 (0.07334423551601243, 'longitude'),
 (0.06290907048262032, 'latitude'),
 (0.056419179181954014, 'rooms_per_hhold'),
 (0.053351077347675815, 'bedrooms_per_room'),
 (0.04114379847872964, 'housing_median_age'),
 (0.014874280890402769, 'population'),
 (0.014672685420543239, 'total_rooms'),
 (0.014257599323407808, 'households'),
 (0.014106483453584104, 'total_bedrooms'),
 (0.010311488326303788, '<1H OCEAN'),
 (0.0028564746373201584, 'NEAR OCEAN'),
 (0.0019604155994780706, 'NEAR BAY'),
 (6.0280386727366e-05, 'ISLAND')]
```

Evaluate Your System on the Test Set

```
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)
```

```
41424.40026462184
```

```
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])
```

Evaluate Your System on the Test Set

- If you did a lot of hyperparameter tuning, the performance on the test set will usually be slightly worse than what you measured using cross-validation.
  - If this happened, DO NOT tweak the hyperparameters to make the numbers look good on the test set.
  - It will not generalize to new data.
- Did we achieve our goal?
  - No, the experts' estimate were usually off by 20%.
  - Is the model useless?

7.
Present Your Solution

Prelaunch Phase

- Present your solution:
  - Highlighting what you have learned
  - What worked and what did not,
  - What assumptions were made
  - What your system’s limitations are
  - Document everything
  - Create presentations with clear visualizations and easy-to-remember statements
    - e.g. the median income is the number one predictor of housing prices

8.
Launch, Monitor, and Maintain Your System

Deploy Your Model

➢ Make a full pipeline with both preparation and prediction:

```
full_pipeline_with_predictor = Pipeline([
        ("preparation", full_pipeline),
        ("linear", LinearRegression())
    ])

full_pipeline_with_predictor.fit(housing, housing_labels)
full_pipeline_with_predictor.predict(some_data)
```

```
array([210644.60459286, 317768.80697211, 210956.43331178,  59218.98886849,
       189747.55849879])
```

Deploy Your Model

- Save the trained Scikit-Learn model:

```
my_model = full_pipeline_with_predictor
```

```
import joblib
joblib.dump(my_model, "my_model.pkl") # DIFF
#...
my_model_loaded = joblib.load("my_model.pkl") # DIFF
```

- Load the trained model within your production environment and use it to make predictions by calling its `predict()` method.

Monitor Your Model

- Write monitoring code to check your system's live performance at regular intervals.
- Trigger alerts when performance drops.
- Models tend to rot over time.
  - The world changes, and if the model was trained with last year's data, it may not be adapted to today's data.
- Put in place:
  - a monitoring system (with or without human raters to evaluate the live model).
  - the relevant processes to define what to do in case of failures and how to prepare for them.

Update the Dataset

- If the data keeps evolving, you will need to update your datasets and retrain your model regularly.

- Automate the whole process as much as possible:
  - Collect fresh data regularly and label it.
  - Write a script to train the model and fine-tune the hyperparameters automatically.
  - Write a script that will evaluate both the new model and the old one on the updated test set, and deploy the model to production if the performance has not decreased.

Maintain Your Model

- Keep backups of every model you create.
  - Have the process and tools in place to roll back to a previous model quickly, in case the new model starts failing badly.

- Keep backups of every version of your datasets.
  - Roll back to a previous dataset if the new one ever gets corrupted.