MYONGJI UNIVERSITY

머신러닝 프로젝트 처음부터 끝까지
- Part 3 -

MYONGJI UNIVERSITY

머신러닝 모델 선택과 훈련

MYONGJI UNIVERSITY

진행 중인 프로젝트 개요

- 목표: 캘리포니아 주 district(구역) 별 중간 주택 가격 예측 모델

- 다음 두가지 사항을 결정해야한다.
  - 주어진 구역의 중간 주택 가격 예측을 위해 어떤 종류의 머신러닝 모델을 사용할 것인지?
  - 머신러닝 모델 성능 측정 지표로 무엇을 사용할 것인지?

- 머신러닝 모델: **회귀 모델**

- 회귀 모델 성능 측정 지표: 평균 제곱근 오차(**RMSE)를 사용**

훈련셋 대상 훈련 및 평가

- 지금까지 진행 상황 정리
  - 훈련셋/테스트셋 구분
  - 변환 파이프라인을 이용한 훈련셋 데이터 전처리 ➔ 훈련셋 준비가 완료된 상황

- 이번에 할 일
  - 예측에 사용할 머신러닝 모델 선택 후 훈련시키기
  - E.g., 선형 회귀(Linear Regression), 결정트리 회귀(Decision Tree Regression), 랜덤 포레스트 회귀(Random Forest Regression) 모델

- 머신러닝 모델 선택 후 사이킷런을 이용하여 모델 훈련시키는 과정은 간단함
  - 전처리 완료된 훈련 데이터셋에 대해서 predictor(예측 모델)의 fit() 메서드 호출

예측 모델 예1: 선형 회귀 모델(Linear Regression)

- 사이킷런의 LinearRegression 클래스를 활용하여 선형 회귀 모델 생성
- 앞서 구현한 데이터 전처리 파이프라인(preprocessing 변환기)에 LinearRegression 예측기 추가

```
from sklearn.linear_model import LinearRegression

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

fit.transform함수는 preprocessing변환기에서 먼저 실행되는 것이고 fit 함수는 다 끝나고 마지막 함수에 대해서 수행한다.

  - housing: 훈련 샘플들에 대한 입력 특성(predictors)
  - housing_labels: 훈련 샘플들에 대한 타깃(레이블) 특성

  - lin_reg 파이프라인의 fit() 함수 호출 시
    - preprocessing 변환기에 대해 fit_transform() 우선 실행된 후(즉, 훈련셋에 대한 전처리 과정 우선 실행됨) 먼저 호출이 되고 실행된다.
    - 전처리 완료된 훈련셋을 이용하여 LinearRegression 예측기 훈련(fit() 실행됨)이 진행됨
  - Q: lin_reg 파이프라인은 estimator, transformer, predictor 중 어느 것에 해당될까?

A) 마지막 객체에 따라서 결정되므로 LinearRegression이므로 predictor에 해당하므로 lin_reg파이프라인도 predictor에 해당함.

예측 모델 예1: 선형 회귀 모델(Linear Regression)

- 훈련된 LinearRegression 예측기를 활용하여 훈련 샘플들에 대한 예측 수행
- 훈련셋의 첫 5개 구역에 대해 예측기가 예측한 가격(housing_predictions)과 실제 가격(housing_labels)을 비교

```
>>> 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.])
```

처음 5개의 훈련결과

각 훈련에 대한 타켓특성값도 확인가능

선형회귀모델을 통해 실제 예측값과 타겟값을 비교한 것

예측 모델 예1: 선형 회귀 모델(Linear Regression)

- RMSE를 성능 측정 지표로 LinearRegression 모델의 훈련 샘플들에 대한 예측 성능 측정
  - 사이킷런의 **root_mean_squared_error 함수**를 이용하여 RMSE 측정
  - RMSE 계산 결과 모델의 예측 성능이 그다지 좋지 않음 예측 오차의 값이므로 값이 커 성능이 좋지 않다 해석가능

```
from sklearn.metrics import root_mean_squared_error

lin_rmse = root_mean_squared_error(housing_labels, housing_predictions)
lin_rmse
```

```
68972.88910758484
```

- 모델이 과소적합(underfitting)된 사례에 해당
  - 과소적합을 해결하는 방법: 예측에 더 유용한 특성을 훈련 데이터에 추가, 모델 훈련 시 적용된 규제를 완화, 또는 더 powerful 한 머신러닝 모델로 변경
  - 위 LinearRegression 모델에는 규제가 전형 적용되지 않았음
  - 과소적합 해결을 위해 결정트리 회귀(Decision Tree Regression) 모델을 시도할 것임

지금 사례에서 발생한 이유는 너무 단순한 선형모델 이였기 때문이다 (3번째 문제)

평균 제곱근 오차(Root Mean Square Error, RMSE)

- 주어진 샘플들에 대한 모델의 예측값과 타깃 간 오차의 제곱의 평균값
- Euclidean norm 또는 ℓ_2 norm 으로 불림

$$\mathrm{RMSE}\left(\mathbf{X}, h\right) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

- 위 수식에 사용된 notation들에 대해 살펴보자.

예측 모델 예2: 결정트리 회귀 모델(Decision Tree Regression)

- 결정트리 모델은 데이터로부터 복잡한 비선형 관계를 학습하는데 보다 효과적임
- 사이킷런의 DecisionTreeRegressor 클래스를 활용하여 결정트리 회귀 모델 생성
- preprocessing 파이프라인에 DecisionTreeRegressor 예측기 추가

```
from sklearn.tree import DecisionTreeRegressor

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

  - housing: 훈련 샘플들에 대한 입력 특성(predictors)
  - housing_labels: 훈련 샘플들에 대한 타깃(레이블) 특성

  - tree_reg 파이프라인의 fit() 메서드 호출을 통해 훈련셋 전처리 과정 이어서 DecisionTreeRegressor 예측기 훈련까지 진행됨

예측 모델 예2: 결정트리 회귀 모델(Decision Tree Regression)

- 훈련된 DecisionTreeRegressor 예측기를 활용하여 훈련 샘플들에 대한 예측 수행
- RMSE를 활용한 DecisionTreeRegressor 모델의 예측 성능 측정
  - 사이킷런의 mean_squared_error 클래스를 이용하여 RMSE 측정

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

```
0.0
```

0이라는 값은 예측값과 정답과 일치한다고 볼 수 있다.

예측 모델 예2: 결정트리 회귀 모델(Decision Tree Regression)

- 훈련된 DecisionTreeRegressor 예측기를 활용하여 훈련 샘플들에 대한 예측 수행
- RMSE를 활용한 DecisionTreeRegressor 모델의 예측 성능 측정
  - 사이킷런의 mean_squared_error 클래스를 이용하여 RMSE 측정

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

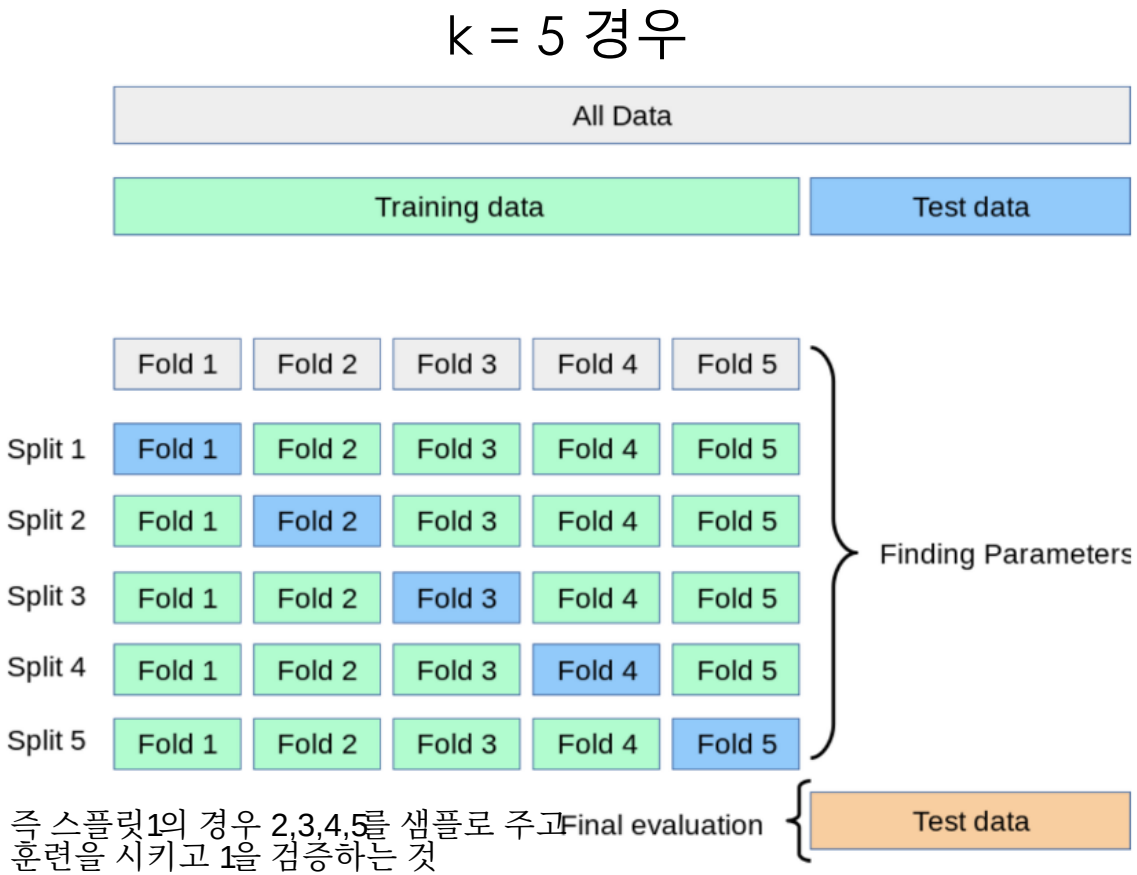
```
0.0
```

- DecisionTreeRegressor 모델 성능 측정 결과
  - **훈련셋에 대한 RMSE 계산 결과가 0**으로 training error가 전혀 없어 보이지만 이는 실제로는 불가능함
  - 모델이 **훈련셋에 심각하게 과대적합(overfitting) 된 사례에 해당**함
  - **교차 검증(cross validation)을 통해 훈련셋에 과대적합 여부 확인 가능**

교차 검증(Cross Validation)

k-fold 교차 검증

- 훈련셋(training data)를 k개의 서브셋으로 랜덤하게 나눔
  - 서브셋을 fold라고 칭함
- 모델 훈련 및 검증 과정을 k번 반복
  - K번 각각에 대해서 서로 다른 1개 폴드를 선택하여 검증 데이터셋(validation set)으로 활용
  - 나머지 (k-1)개 폴드를 활용하여 모델 훈련 후 선택된 검증 용 폴드를 이용하여 모델 성능 측정(e.g., RMSE 측정)
- 최종적으로 k개의 성능 측정 결과치가 저장된 배열 생성됨



결정트리 회귀 모델 교차 검증 예(k = 10)

```
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)
```

입력데이터, 타켓데이터를 파라메타로 받음.

- K-fold 교차 검증을 위해서는 사이킷런의 **cross_val_score() 함수 활용**
- cross_val_score() 호출 시 scoring 매개변수를 통해 사용할 성능 측정 지표를 지정해야함.
  - **scoring 매개변수 값으로 성능 측정 지표 값이 클 수록 더 좋은 성능을 의미하는 효용 함수 (utility function)를 기대**하지만, RMSE는 값이 작을 수록 더 좋은 성능을 의미하는 비용 함수 (cost function)에 해당한다. RMSE는 값이 클수록 오차가 크다는 것을 뜻했으므로 반대개념이다. 따라서 음수로 저장.
  - 이를 위해 **RMSE의 음수값으로 scoring 매개변수를 지정**했으며 **cross_val_score() 리턴값에 대해서도 음수를 취했음**
  - 이 외에도 교차 검증 대상 모델의 종류에 따라 다양한 지표를 사용하여 모델의 성능을 측정할 수 있으며 scoring 매개변수를 통해 지정함

결정트리 회귀 모델 교차 검증 예(k = 10)

```
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)
```

- 10-fold cross validation 실행 결과로 10개 성능 평가 결과치가 리턴됨

```
>>> 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
```

Training error는 0이었던데 반해 교차 검증으로 확인된 mean RMSE는 66868이므로 과대적합 되었음이 확인됨

앙상블(Ensemble) 기법

- 결정트리(decision tree) 모델 하나보다 **여러 결정트리 모델로 이루어진 random forest 모델이 일반적으로 보다 좋은 성능**을 낸다.

- 이처럼 **여러 개별 모델들을 함께 구성하여(ensemble) 학습시킨 후 각 모델의 예측값들의 평균값을 사용**하면 보다 좋은 성능을 내는 모델을 얻을 수 있다.

랜덤 포레스트 회귀 모델(Random Forest Regression) 및 교차 검증

```
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)
```

- 랜덤 포레스트 회귀 모델은 여러 개의 결정트리들로 구성됨
  - 이와 같이 여러 개의 개별 모델들을 모아서 하나의 모델을 구성하는 기법을 앙상블이라고 부름
- 각 결정트리는 주어진 모든 특성(feature)들의 랜덤 서브셋으로 훈련됨
- 최종 예측값으로는 각 결정트리 모델의 예측값들의 평균을 취함

10개의 평균값

```
>>> 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
```

랜덤포레스트로 한것이 선형회귀보다 더 오차가 적은것을 볼 수 있음.

모델 세부 튜닝 (하이퍼파라미터 튜닝)

가능성 있는 모델을 추렸다면 이제 추려진 모델을 세부 튜닝해야함

하이퍼파라미터 튜닝

- 지금까지 살펴 본 모델 중 랜덤 포레스트 회귀 모델의 성능이 가장 우수했음
- 가능성이 높은 모델 선정 후에는 모델 훈련 과정(학습 알고리즘)에 대한 세부 설정(하이퍼파라미터)를 튜닝해야함

- 하이퍼파라미터 튜닝을 위한 2가지 방식
  - **그리드 탐색(Grid search)**
  - **랜덤 탐색(Randomized search)**

그리드 탐색(Grid Search)

```
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)
```

사이킷런의 **GridSearchCV 클래스** 활용

- 랜덤 포레스트 회귀 모델(random_forest)에 대해 그리드 탐색을 통한 하이퍼파라미터 튜닝을 진행
- 튜닝하고자 하는 하이퍼파라미터들을 선정
  - **preprocessing__geo__n_clusters** : preprocessing 파이프라인에 포함된 "geo"라고 이름 붙여진 ClusterSimilarity 변환기의 n_clusters 하이퍼파라미터를 의미 몇개의 샘플 군집을 설정한것인지 대한 것
  - **random_forest__max_features** : random_forest regressor의 max_features 하이퍼파라미터

앞서 구현했던 preprocessing 변환 파이프라인

```
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
```

그리드 탐색(Grid Search)

```
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)
```

사이킷런의 GridSearchCV 클래스 활용

여긴 두세트로 나눠서 시도

- 튜닝 대상 각 하이퍼파라미터에 대해 테스트 하고자 하는 값들의 리스트를 지정함
  - {'preprocessing__geo__n_clusters' : [5, 8, 10], 'random_forest__max_features' : [4,6,8]}
- 지정된 사항을 토대로 모든 가능한 하이퍼파라미터 값 조합에 대해 교차 검증을 진행하여 최적의 하이퍼파라미터 값 조합을 찾는다.
  - 위 param_grid에 지정된 내용에 따라 교차 검증할 모든 하이퍼파라미터 값 조합의 개수는 (3x3)+(2x3) = 15
  - 또한 cv=3(3-fold 교차 검증) 이므로 15가지 각 케이스에 대해 3번씩 훈련 및 검증 진행

랜덤 포레스트 회귀 모델에 대한 그리드 탐색 결과

Colab 확인

- 그리드 탐색 결과 최적의 하이퍼파라미터 조합

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

- 상위 5개 하이퍼파라미터 조합

```
>>> 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()  # note: the 1st column is the row ID
   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
```

랜덤 탐색(Randomized search)

- 그리드 탐색은 탐색하고자 하는 하이퍼파라미터 값 조합들의 경우의 수가 적은 경우에 적합

- 반면에 **탐색하고자 하는 하이퍼파라미터 값 조합들의 경우의 수가 굉장히 많은 경우**에는 그리드 탐색이 부적합하며 **랜덤 탐색이 효율적**
  - E.g., 탐색하고자 하는 하이퍼파라미터 수가 6개이고 각 하이퍼파라미터에 대해 가능한 값들의 수가 10개인 경우 모든 가능한 하이퍼파라미터 조합들의 개수는 10^6 임. 10^6개 경우에 대해 교차 검증을 수행하는 것은 상당히 비효율적임.
  - E.g., 탐색하고자 하는 하이퍼파라미터가 continuous 변수인 경우

- 사이킷런의 RandomizedSearchCV 클래스 활용

랜덤 포레스트 회귀 모델에 대한 랜덤 탐색

```
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)
```

- **param_distribs**: **탐색할 하이퍼파라미터** 및 **각 하이퍼파라미터 값의 분포**를 명시
- RandomizedSearchCV(..)
  - **n_iter=10**: 총 10가지 하이퍼파라미터 조합을 테스트. 각 케이스마다 n_clusters와 max_features 값을 명시된 범위에서 랜덤하게 선택.
  - cv=3: 10가지 케이스 각각에 대해 3-fold 교차 검증 진행 ➔ 훈련 및 검증 과정이 (10*3)=30번 반복 진행됨.

랜덤 포레스트 회귀 모델에 대한 랜덤 탐색 결과

```
# extra code - displays the random search results
cv_res = pd.DataFrame(rnd_search.cv_results_)
cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)
cv_res = cv_res[["param_preprocessing__geo__n_clusters",
                 "param_random_forest__max_features", "split0_test_score",
                 "split1_test_score", "split2_test_score", "mean_test_score"]]
cv_res.columns = ["n_clusters", "max_features"] + score_cols
cv_res[score_cols] = -cv_res[score_cols].round().astype(np.int64)
```

cv_res

	n_clusters	max_features	split0	split1	split2	mean_test_rmse
1	45	9	41287	42150	42627	42021
8	32	7	41690	42542	43224	42485
0	41	16	42223	42959	43321	42834
5	42	4	41818	43094	43817	42910
2	23	8	42264	42996	43830	43030
6	24	3	42693	43421	44441	43518
7	26	13	43097	43521	44175	43598
3	21	12	43721	43915	44543	44060
4	13	5	43902	43710	45087	44233
9	4	2	50364	49740	51560	50555

<하이퍼파라미터 튜닝 전 10-fold 교차검증 결과>

```
>>> 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
```

MYONGJI UNIVERSITY

최상의 모델 분석

Colab 확인

- 또한 그리드 또는 랜덤 탐색을 통해 얻어진 best 모델을 분석하면 문제 해결에 대한 좋은 insight를 얻는 경우가 많음
- E.g., 랜덤 탐색을 통해 찾은 best 랜덤 포레스트 회귀 모델에서 주택 가격 예측에 사용된 각 특성의 상대적인 중요도 확인 가능

```
>>> 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])
>>> 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'),
 (0.05446998753775219, 'rooms_per_house__ratio'),
 (0.05262301809680712, 'people_per_house__ratio'),
 (0.03819415873915732, 'geo__Cluster 0 similarity'),
 [...]
 (0.00015061247730531558, 'cat__ocean_proximity_NEAR BAY'),
 (7.301686597099842e-05, 'cat__ocean_proximity_ISLAND')]
```

- median_incom과 ocean_proximity_INLAND가 가장 유용한 특성이며, ocean_proximity 관련 나머지 4가지 특성은 중요도가 낮음
- 중요도가 낮은 특성은 데이터셋으로부터 제외시킬 수 있음

테스트셋을 이용하여 모델 최종 평가

```
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
```

- 랜덤 탐색을 통해 찾은 best 모델 사용
  - final_model = rnd_search.best_estimator_ # includes preprocessing
- 테스트셋으로부터 예측에 활용할 특성들(입력특성 or predictors)과 타깃(레이블)을 분리
  - X_test(예측에 활용할 특성들), y_test(타깃)
- final_model.predict(X_test)
  - preprocessing 변환 파이프라인을 통한 X_test에 대한 전처리가 이루어짐
  - 이어서 전처리 된 X_test를 토대로 주택 가격 예측이 진행됨
  - final_predictions가 X_test에 포함된 각 샘플(district)에 대한 랜덤 포레스트 회귀 모델의 중간 주택 가격 예측 결과임
- mean_squared_error() 함수를 이용하여 y_test(타깃(레이블))과 final_predictions(예측 결과) 간 RMSE를 계산

테스트셋으로 모델 평가

Colab 확인

- 계산된 일반화 오차(generalization error) 추정치가 얼마나 정확한지 확인하기 위해 추가로 일반화 오차에 대한 95% 신뢰 구간 계산

```
>>> 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])
```

final_rmse= 41424.400…

론칭, 모니터링, 시스템 유지 보수

실제 시스템에 모델 론칭

- 하이퍼파라미터 튜닝 및 훈련을 마친 최고 성능 모델(전처리 파이프라인 및 RandomForestRegressor 예측기 포함)을 파일로 저장
  - joblib 라이브러리 사용하여 가능
- 저장된 파일을 production 환경으로 옮긴 다음 파일로부터 모델을 로드해서 시스템에 적용

실제 시스템에 모델 론칭

- E.g., 웹 서비스 형태로 모델 배포
  - 캘리포니아 주 district의 주택 가격 예측 서비스를 제공하는 웹 사이트를 가정
  - 사용자는 웹 사이트에서 관심 있는 특정 district에 관한 정보를 입력한 후 "가격 예측" 버튼을 클릭
  - 사용자가 입력한 정보가 웹 서버로 전송되어 웹 어플리케이션에 전달됨
  - 웹 어플리케이션은 REST API를 통해 사용자가 입력한 정보를 웹 서비스(RandomForestRegressor모델)로 전달하고 모델은 예측 결과값을 reply
  - 장점: 확장이 용이

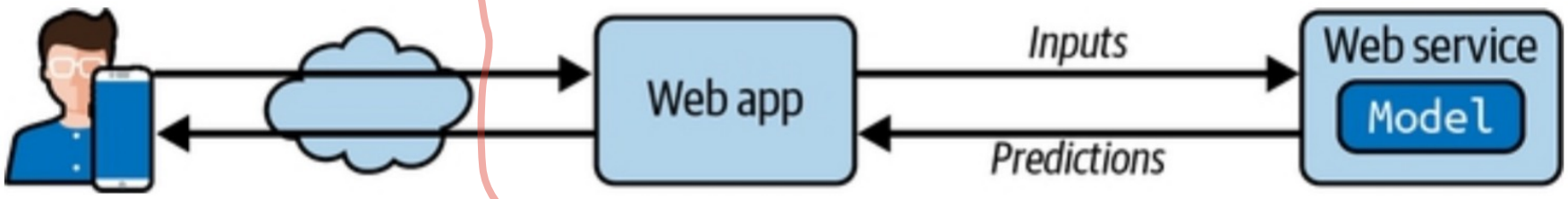


Figure 2-20. A model deployed as a web service and used by a web application

클라우드를 통한 모델 론칭

- E.g., Google의 Vertex AI 같은 클라우드를 통해 머신러닝 모델 배포 가능
  - joblib 라이브러리를 사용하여 저장된 모델 파일을 Google Cloud Storage(GCS)에 업로드
  - Vertex AI에서 새로운 모델 버전을 생성한 다음 GCS에 업로드 한 모델 파일을 지정해줌
  - Vertex AI에서 모델에 대한 웹 서비스를 생성해주고 확장 및 로드 밸런싱 등을 알아서 처리해줌

성능 모니터링을 위한 시스템 구축

- 다양한 원인으로 머신러닝 시스템 성능 저하가 발생할 수 있음
  - E.g., 시스템을 구성하는 컴포넌트 고장으로 인한 갑작스런 성능 저하, 긴 시간에 걸쳐 (눈에 뜨지 않는) 점진적인 성능 저하, 시간이 지남에 따라 모델 낙후로 인한 성능 저하
- 시스템 성능 저하를 감지하기 위한 모니터링 시스템 필요
- E.g., 머신러닝 모델 기반의 제품 추천 시스템에 대한 모니터링
  - 머신러닝 모델에 의해 추천된 상품의 판매량 모니터링
- E.g., 생산라인에서 제품에 대한 이미지 분류 모델 모니터링
  - 모델이 분류한 전체 이미지 중 일부 샘플을 사람 평가자가 다시 한번 확인
  - 작업의 종류에 따라 사람 평가자는 전문가이거나 crowdsourcing 플랫폼(e.g., Amazon Mechanical Turk)의 작업자 같은 비전문가일 수 있음
- 모델의 예측 성능이 일정 수준 이하로 저하될 경우 어떻게 대비할 것인지에 관한 대비 프로세스 준비 또한 필요함

데이터 변화에 따른 모델 업데이트

- 시간이 지남에 따라 진화하는 성격의 데이터라면 정기적으로 데이터셋을 업데이트하고 업데이트 된 데이터셋으로 모델을 다시 훈련해야함

- 필요한 작업 및 자동화 각 샘플(타겟 특성)에 대한 레이블링 작업도 필요
  - 정기적으로 새로운 데이터를 수집하고 레이블링 함(e.g., 사람 조사원이 레이블링)
  - 데이터 전처리, 모델 재훈련 및 하이퍼파라미터 튜닝 과정을 자동화 하기 위한 파이프라인 구현
  - 업데이트 된 테스트셋으로 새로운 모델과 기존 모델을 평가하는 스크립트 작성 ➔ 새로운 모델의 성능이 더 나쁘지 않다면 새로운 모델로 교체

데이터셋 및 모델 백업

- **모델 백업**: 새로운 모델이 문제가 있을 경우 이전 모델로 신속하게 rollback 할 수 있도록 훈련을 마친 완성된 모델은 항상 백업해두는 것이 바람직함

- **데이터셋 백업**: 새로운 버전의 데이터셋이 어떤 이유로 오염(e.g., 이상치가 굉장히 많음)되었다면 rollback 할 수 있도록 모든 버전의 데이터셋 또한 백업하는 것이 바람직함