2장 머신러닝 프로젝트 처음부터 끝까지 (2부)

감사의 글

자료를 공개한 저자 오렐리앙 제롱과 강의자료를 지원한 한빛아카데미에게 진심어린 감사를 전합니다.

2.5 머신러닝 알고리즘을 위한 데이터 준비

데이터 준비 자동화

- 모든 전처리 과정의 자동화. 언제든지 재활용 가능.
- 자동화는 **파이프라인**(pipeline)으로 구현
- 훈련 세트 준비: 훈련에 사용되는 특성과 타깃 특성(레이블) 구분하여 복사본 생성

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

- 테스트 세트는 훈련이 완성된 후에 성능 측정 용도로만 사용.

데이터 전처리

- 데이터 전처리(data preprocessing): 효율적인 모델 훈련을 위한 데이터 변환
- 수치형 데이터와 범주형 데이터에 대해 다른 변환과정을 사용
- 수치형 데이터 전처리 과정
    - 데이터 정제
    - 조합 특성 추가
    - 특성 스케일링
- 범주형 데이터 전처리 과정
    - 원-핫-인코딩(one-hot-encoding)

변환 파이프라인

- **파이프라인**(pipeline)
    - 여러 과정을 한 번에 수행하는 기능을 지원하는 도구
    - 여러 사이킷런 API를 묶어 순차적으로 처리하는 사이킷런 API

- 파이프라인 적용 과정
    - 수치형 데이터 전처리 과정에 사용된 세 가지 변환 과정의 자동화 파이프라인 구현
    - 수치형 데이터 파이프라인과 범주형 데이터 전처리 과정을 결합한 파이프라인 구현

사이킷런 API 활용

- '조합 특성 추가' 과정을 제외한 나머지 변환은 사이킷런에서 제공하는 관련 API 직접 활용 가능
- '조합 특성 추가' 과정도 다른 사이킷런 API와 호환이 되는 방식으로 사용자가 직접 구현 가능
- 사이킷런에서 제공하는 API는 일관되고 단순한 인터페이스를 제공

사이킷런 API의 세 가지 유형

추정기(estimator)

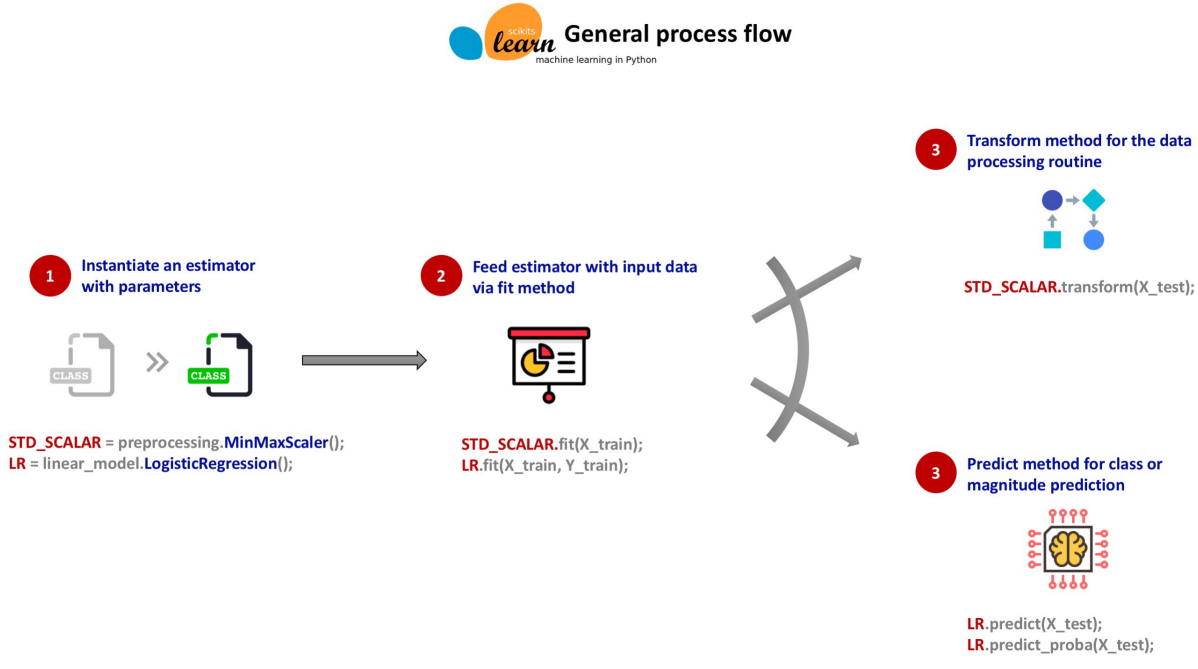
- 주어진 데이터셋과 관련된 특정 파라미터 값들을 추정하는 객체
- `fit()` 메서드 활용: 특정 파라미터 값을 저장한 속성이 업데이트된 객체 자신 반환

변환기(transformer):

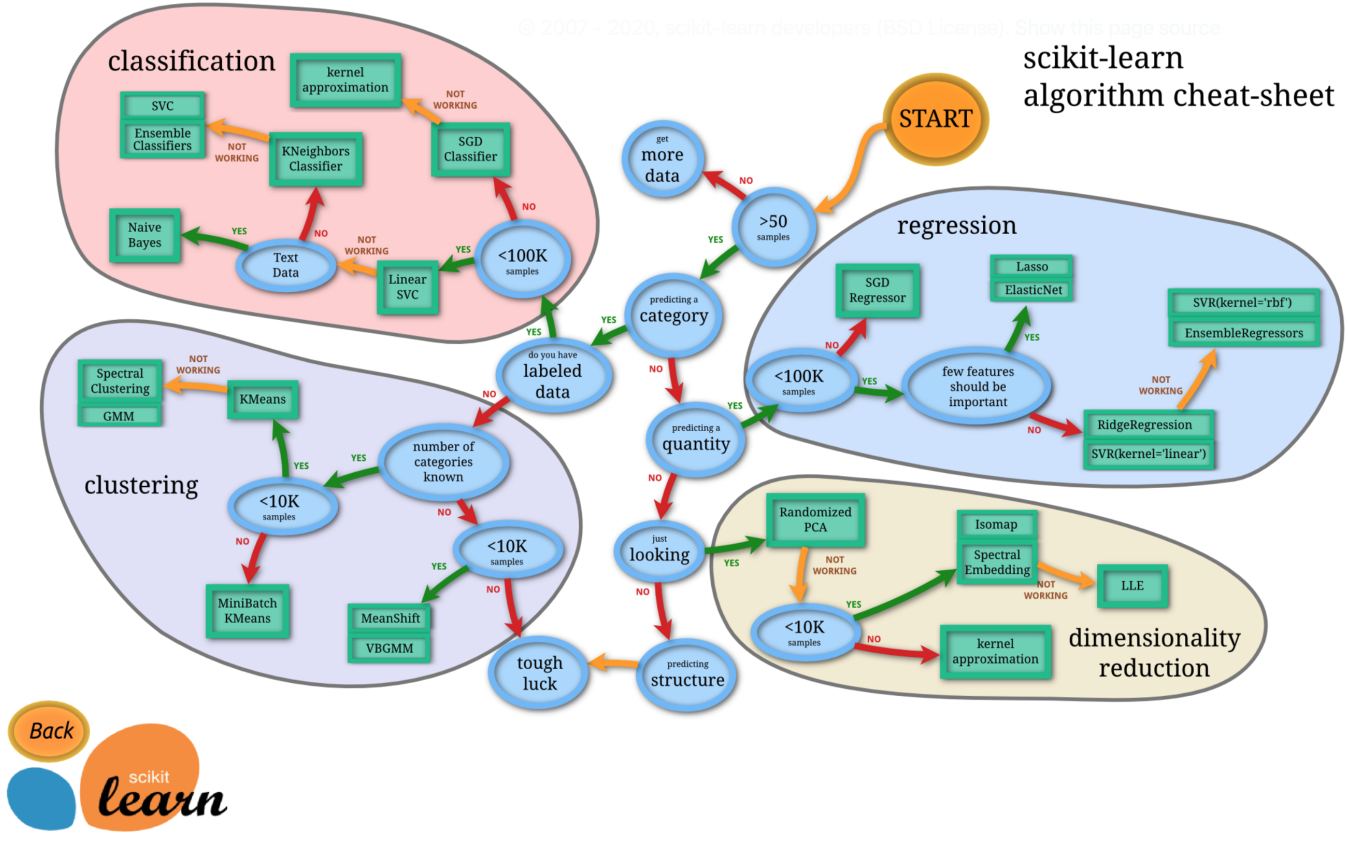
- fit() 메서드에 의해 학습된 파라미터를 이용하여 주어진 데이터셋 변환
- `transform()` 메서드 활용
- `fit()` 메서드와 `transform()` 메서드를 연속해서 호출하는 `fit_transform()` 메서드 활용 가능

예측기(predictor)

- 주어진 데이터셋과 관련된 값을 예측하는 기능을 제공하는 추정기
- `predict()` 메서드 활용
- `fit()`과 `predict()` 메서드가 포함되어 있어야 함
- `predict()` 메서드가 추정한 값의 성능을 측정하는 `score()` 메서드도 포함
- 일부 예측기는 추정치의 신뢰도를 평가하는 기능도 제공



<그림 출처: [Scikit-Learn: A silver bullet for basic machine learning (https://medium.com/analytics-vidhya/scikit-learn-a-silver-bullet-for-basic-machine-learning-13c7d8b248ee)](https://medium.com/analytics-vidhya/scikit-learn-a-silver-bullet-for-basic-machine-learning-13c7d8b248ee)>



<그림 출처: [Scikit-Learn: Choosing the right estimator (https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html)](https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html)>

2.5.1 데이터 정제: 수치형 데이터 전치러 과정 1

- 누락된 특성값이 존재 경우, 해당 값 또는 특성을 먼저 처리해야 함.

- `total_bedrooms` 특성에 207개 구역에 대한 값이 null로 채워져 있음, 즉, 일부 구역에 대한 정보가 누락됨.

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_p
4629	-118.30	34.07	18.0	3759.0	NaN	3296.0	1462.0	2.2708	<1H
6068	-117.86	34.01	16.0	4632.0	NaN	3038.0	727.0	5.1762	<1H
17923	-121.97	37.35	30.0	1955.0	NaN	999.0	386.0	4.6328	<1H
13656	-117.30	34.05	6.0	2155.0	NaN	1039.0	391.0	1.6675	
19252	-122.79	38.48	7.0	6837.0	NaN	3468.0	1405.0	3.1662	<1H

null 값 처리 옵션

- 옵션 1: 해당 구역 제거
- 옵션 2: 전체 특성 삭제
- 옵션 3: 평균값, 중앙값, 0, 주변에 위치한 값 등 특정 값으로 채우기. 책에서는 중앙값으로 채움.

옵션	코드
옵션 1	`housing.dropna(subset=["total_bedrooms"])`
옵션 2	`housing.drop("total_bedrooms", axis=1)`
옵션 3	`median = housing["total_bedrooms"].median()`
	`housing["total_bedrooms"].fillna(median, inplace=True)`

<옵션 3 활용>

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_p
4629	-118.30	34.07	18.0	3759.0	433.0	3296.0	1462.0	2.2708	<1H
6068	-117.86	34.01	16.0	4632.0	433.0	3038.0	727.0	5.1762	<1H
17923	-121.97	37.35	30.0	1955.0	433.0	999.0	386.0	4.6328	<1H
13656	-117.30	34.05	6.0	2155.0	433.0	1039.0	391.0	1.6675	
19252	-122.79	38.48	7.0	6837.0	433.0	3468.0	1405.0	3.1662	<1H

SimpleImputer 변환기

- 옵션 3를 지원하는 사이킷런 변환기
- 중앙값 등 통계 요소를 활용하여 누락 데이터를 특정 값으로 채움

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
housing_num = housing.drop("ocean_proximity", axis=1)
imputer.fit(housing_num)
```

2.5.2 텍스트와 범주형 특성 다루기: 원-핫 인코딩

- 범주형 특성인 해안 근접도(ocean_proximity)를 수치형 데이터로 변환해야 함.

- 단순 수치화 적용 가능

범주	숫자
<1H OCEAN	0
INLAND	1
ISLAND	2
NEAR BAY	3
NEAR OCEAN	4

단순 수치화의 문제점

- 해안 근접도는 단순히 구분을 위해 사용. 해안에 근접하고 있다 해서 주택 가격이 기본적으로 더 비싸지 않음.

- 반면에 수치화된 값들은 크기를 비교할 수 있는 숫자

- 따라서 모델 학습 과정에서 숫자들의 크기 때문에 잘못된 학습이 이루어질 수 있음.

원-핫 인코딩(one-hot encoding)

- 수치화된 범주들 사이의 크기 비교를 피하기 위해 더미(dummy) 특성을 추가하여 활용
    - 범주 수 만큼의 더미 특성 추가

- 예를 들어, 해안 근접도 특성 대신에 다섯 개의 범주 전부를 새로운 특성으로 추가한 후 각각의 특성값을 아래처럼 지정
    - 해당 카테고리의 특성값: 1
    - 나머지 카테고리의 특성값: 0

OneHotEncoder 변환기

- 원-핫 인코딩 지원

- `sparse` 키워드 인자
  - 기본값은 `True`.
  - 1의 위치만 기억하는 희소 행렬로 처리. 대용량 행렬 처리에 효과적임.
  - `False`로 지정할 경우 일반 행렬로 처리.

```
In [68]: cat_encoder = OneHotEncoder(sparse=False)
         housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
         housing_cat_1hot

Out[68]: array([[1., 0., 0., 0., 0.],
                [1., 0., 0., 0., 0.],
                [0., 0., 0., 0., 1.],
                ...,
                [0., 1., 0., 0., 0.],
                [1., 0., 0., 0., 0.],
                [0., 0., 0., 1., 0.]])
```

2.5.3 나만의 변환기: 수치형 데이터 전처리 과정 2 (조합 특성 추가)

- 아래 특성 추가 용도 변환기 클래스 직접 선언하기
    - 가구당 방 개수(rooms for household)
    - 방 하나당 침실 개수(bedrooms for room)
    - 가구당 인원(population per household)

- 변환기 클래스: `fit()`, `transform()` 메서드를 구현하면 됨.
    - 주의: fit() 메서드의 리턴값은 self

예제: CombinedAttributesAdder 변환기 클래스 선언

- `__init__()` 메서드: 생성되는 모델의 **하이퍼파라미터** 지정 용도
  - 모델에 대한 적절한 하이퍼파라미터를 튜닝할 때 유용하게 활용됨.
  - 예제: 방 하나당 침실 개수 속성 추가 여부
- `fit()` 메서드: 계산해야 하는 파라미터가 없음. 바로 `self` 리턴
- `transform()` 메서드: 넘파이 어레이를 입력받아 속성을 추가한 어레이를 반환

```
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True):
        ...

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        ...
```

상속하면 좋은 클래스

- `BaseEstimator` 상속: 하이퍼파라미터 튜닝 자동화에 필요한 `get_params()`, `set_params()` 메서드 제공

- `TransformerMixin` 상속: `fit_transform()` 자동 생성

2.5.4 특성 스케일링: 수치형 데이터 전처리 과정 3

- 머신러닝 알고리즘은 입력 데이터셋의 특성값들의 스케일(범위)이 다르면 제대로 작동하지 않음
- 특성에 따라 다루는 숫자의 크기가 다를 때 통일된 스케일링이 필요
- 아래 두 가지 방식이 일반적으로 사용됨.
    - min-max 스케일링
    - 표준화 (책에서 사용)
- **주의**: 타깃(레이블)에 대한 스케일링은 하지 않음

min-max 스케일링

- **정규화**(normalization)라고도 불림
- 특성값 x를 $\frac{x-min}{max-min}$로 변환
- 변환 결과: **0에서 1** 사이
- 이상치에 매우 민감
    - 이상치가 매우 **크면 분모가 매우 커져서** 변환된 값이 **0 근처**에 몰림

표준화(standardization)

- 특성값 x 를 $\frac{x-\mu}{\sigma}$ 로 변환
    - μ: 특성값들의 **평균**값
    - σ: 특성값들의 **표준편차**

- 결과: 변환된 데이터들이 **표준정규분포**를 이룸
    - 이상치에 상대적으로 영향을 덜 받음.

- 사이킷런의 `StandardScaler` 변환기 활용 가능 (책에서 사용)

변환기 관련 주의사항

- `fit()` 메서드: 훈련 세트에 대해서만 적용. 테스트 세트는 활용하지 않음.

- `transform()` 메서드: 테스트 세트 포함 모든 데이터에 적용
    - 훈련 세트를 이용하여 필요한 파라미터를 확인한 후 그 값들을 이용하여 전체 데이터셋트를 변환

2.5.5 변환 파이프라인

- 모든 전처리 단계가 정확한 순서대로 진행되어야 함
- 사이킷런의 `Pipeline` 클래스를 이용하여 파이프라인 변환기 객체 생성 가능

수치형 데이터 변환 파이프라인

```
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])
```

- 인자: 이름과 추정기로 이루어진 쌍들의 리스트

- 마지막 추정기 제외 나머지 추정기는 모두 변환기이어야 함.
    - `fit_transform()` 메서드 지원

- 파이프라인으로 정의된 추정기의 유형은 마지막 추정기의 유형과 동일
    - `num_pipeline`는 변환기. 이유는 `std_scaler`가 변환기이기 때문임.

- `num_pipeline.fit()` 호출:
    - 마지막 단계 이전 추정기: `fit_transform()` 메소드 연속 호출. 즉, 변환기가 실행될 때마다 변환도 동시에 진행.
    - 마지막 추정기: `fit()` 메서드 호출

수치형 / 범주형 데이터 전처리 과정 통합 파이프라인

- 사이킷런의 `ColumnTransformer` 클래스를 이용하여 특성별로 지정된 전처리를 처리할 수 있도록 지정 가능

- 인자: (이름, 추정기, 적용 대상 열(column) 리스트) 튜플로 이루어진 리스트

- `fit()` 메서드에 pandas의 데이터프레임을 직접 인자로 사용 가능

- 수치형 특성: `num_pipeline` 변환기
  - 적용 대상 열(columns): `list(housing_num)`

- 범주형 특성: `OneHotEncoder` 변환기
  - 적용 대상 열(columns): `["ocean_proximity"]`

```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

2.6 모델 선택과 훈련

- 목표 달성에 필요한 두 요소를 결정해야함
    - 학습 모델
    - 회귀 모델 성능 측정 지표

- 목표: 구역별 중간 주택 가격 예측 모델

- 학습 모델: 회귀 모델

- 회귀 모델 성능 측정 지표: 평균 제곱근 오차(RMSE)

2.6.1 훈련 세트에서 훈련하고 평가하기

- 지금까지 한 일
    - 훈련 세트 / 테스트 세트 구분
    - 변환 파이프라인을 활용한 데이터 전처리

- 이제 할 일
    - 예측기 모델 선택 후 훈련시키기
    - 예제: 선형 회귀, 결정트리 회귀

- 예측기 모델 선택 후 훈련과정은 매우 단순함.
    - `fit()` 메서드를 전처리 처리가 된 훈련 데이터셋에 적용

선형 회귀 모델(4장)

- 선형 회귀 모델 생성: 사이킷런의 **`LinearRegression`** 클래스 활용

- 훈련 및 예측
    - 예측은 훈련 세트에 포함된 몇 개 데이터를 대상으로 예측 실행

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

lin_reg.predict(housing_prepared)
```

선형 회귀 모델의 훈련 세트 대상 예측 성능

- RMSE(평균 제곱근 오차)가 68628.198 정도로 별로 좋지 않음.

- 훈련된 모델이 훈련 세트에 **과소적합** 됨.
    - 보다 좋은 특성을 찾거나 더 강력한 모델을 적용해야 함.
    - 보다 좋은 특성 예제: 로그 함수를 적용한 인구수 등
    - 모델에 사용되는 규제(regulaization, 4장)를 완화할 수도 있지만 위 모델에선 어떤 규제도 적용하지 않았음.

결정트리 회귀 모델(6장)

- 결정 트리 모델은 데이터에서 복잡한 비선형 관계를 학습할 때 사용
- 결정트리 회귀 모델 생성: 사이킷런의 **`DecisionTreeRegressor`** 클래스 활용
- 훈련 및 예측
    - 예측은 훈련 세트에 포함된 몇 개 데이터를 대상으로 예측 실행

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)

housing_predictions = tree_reg.predict(housing_prepared)
```

결정트리 회귀 모델의 훈련 세트 대상 예측 성능

- RMSE(평균 제곱근 오차)가 0으로 완벽해 보임.

- 훈련된 모델이 훈련 세트에 심각하게 **과대적합** 됨.
    - 실전 상황에서 RMSE가 0이 되는 것은 불가능.
    - 훈련 세트가 아닌 테스트 세트에 적용할 경우 RMSE가 크게 나올 것임.

2.6.2 교차 검증을 사용한 평가

- 테스트 세트를 사용하지 않으면서 훈련 과정을 평가할 수 있음.

- **교차 검증** 활용

k-겹 교차 검증

- 훈련 세트를 **폴드**(fold)라 불리는 k-개의 부분 집합으로 무작위로 분할

- 총 k 번 지정된 모델을 훈련
    - 훈련할 때마다 매번 다른 하나의 폴드 선택
    - 다른 (k-1) 개의 폴드를 이용해 훈련
    - 평가는 선택된 폴드 활용

- 최종적으로 k 번의 평가 결과가 담긴 배열 생성

예제: 결정 트리 모델 교차 검증 (k = 10인 경우)

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)

tree_rmse_scores = np.sqrt(-scores)
```

- k-겹 교차 검증의 모델 학습 과정에서 성능을 측정할 때 높을 수록 좋은 **효용함수** 활용
  - `scoring="neg_mean_squared_error"`
  - RMSE의 음숫값

- 교차 검증의 RMSE: 다시 음숫값(`-scores`) 사용
  - 평균 RMSE: 약 71407
  - 별로 좋지 않음.

예제: 선형 회귀 모델 교차 검증 (k = 10 인 경우)

```
lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
                             scoring="neg_mean_squared_error", cv=10)

lin_rmse_scores = np.sqrt(-lin_scores)
```

- 교차 검증의 RMSE 평균: 약 69052
  - 결정트리 회귀 모델보다 좋음.

앙상블 학습 (7장)

- 여러 개의 다른 모델을 모아서 하나의 모델을 만드는 기법
- 머신러닝 알고리즘의 성능을 극대화는 방법 중 하나

랜덤 포레스트 회귀 모델 (7장)

- 앙상블 학습에 사용되는 하나의 기법

- 특성을 무작위로 선택해서 생성한 여러 개의 결정 트리를 훈련 시킨 후 각 모델의 예측 값의 평균 값을 예측값으로 사용하는 모델

- 사이킷런의 `RandomForestRegressor` 클래스 활용

- 훈련 및 예측

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)

housing_predictions = forest_reg.predict(housing_prepared)
```

- 랜덤 포레스트 모델의 RMSE: 약 50182
    - 지금까지 사용해본 모델 중 최고
    - 하지만 여전히 과대적합되어 있음.

2.7 모델 세부 튜닝

- 살펴 본 모델 중에서 **랜덤 포레스트** 모델의 성능이 가장 좋았음

- 가능성이 높은 모델을 선정한 후에 **모델 세부 설정을 튜닝**해야함

- 튜닝을 위한 세 가지 방식
    - **그리드 탐색**
    - **랜덤 탐색**
    - **앙상블 방법**

2.7.1 그리드 탐색

- 지정한 하이퍼파라미터의 모든 조합을 교차검증하여 최선의 하이퍼파라미터 조합 찾기
- 사이킷런의 `GridSearchCV` 활용

예제: 그리드 탐색으로 랜덤 포레스트 모델에 대한 최적 조합 찾기

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor(random_state=42)
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)
```

- 총 (3x4 + 2x3 = 18) 가지의 경우 확인
- 5-겹 교차검증(cv=5)이므로, 총 (18x5 = 90)번 훈련함.

그리드 탐색 결과

- 최고 성능의 랜덤 포레스트 하이퍼파라미터가 다음과 같음.
    - `max_features`: 8
    - `n_estimators`: 30
    - 지정된 구간의 최고값들이기에 구간을 좀 더 넓히는 게 좋아 보임

- 최고 성능의 랜덤 포레스트에 대한 교차검증 RMSE: 49682
    - 하나의 랜덤 포레스트보다 좀 더 좋아졌음.

2.7.2 랜덤 탐색

- 그리드 탐색은 적은 수의 조합을 실험해볼 때 유용

- 조합의 수가 커지거나, 설정된 탐색 공간이 커지면 랜덤 탐색이 효율적
    - 설정값이 연속적인 값을 다루는 경우 랜덤 탐색이 유용

- 사이킷런의 `RandomizedSearchCV` 추정기가 랜덤 탐색을 지원

예제: 랜덤 탐색으로 랜덤 포레스트 모델에 대한 최적 조합 찾기

```
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)
```

- `n_iter=10`: 랜덤 탐색이 총 10회 진행
  - `n_estimators`와 `max_features` 값을 지정된 구간에서 무작위 선택

- `cv=5`: 5-겹 교차검증. 따라서 랜덤 포레스트 학습이 (10x5=50)번 이루어짐.

랜덤 탐색 결과

- 최고 성능의 랜덤 포레스트 하이퍼파라미터가 다음과 같음.
    - `max_features`: 7
    - `n_estimators`: 180

- 최고 성능의 랜덤 포레스트에 대한 교차검증 RMSE: 49150

2.7.3 앙상블 방법

- 결정 트리 모델 하나보다 랜덤 포레스트처럼 여러 모델로 이루어진 모델이 보다 좋은 성능을 낼 수 있음.

- 또한 최고 성능을 보이는 서로 다른 개별 모델을 조합하면 보다 좋은 성능을 얻을 수 있음

- 7장에서 자세히 다룸

2.7.4 최상의 모델과 오차 분석

- 그리드 탐색과 랜덤 탐색 등을 통해 얻어진 최상의 모델을 분석해서 문제에 대한 좋은 통창을 얻을 수 있음

- 예를 들어, 최상의 랜덤 포레스트 모델에서 사용된 특성들의 중요도를 확인하여 일부 특성을 제외할 수 있음.
    - 중간 소득(median income)과 INLAND(내륙, 해안 근접도)가 가장 중요한 특성으로 확인됨
    - 해안 근접도의 다른 네 가지 특성은 별로 중요하지 않음

```
[(0.36615898061813423, 'median_income'),
 (0.16478099356159054, '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')]
```

2.7.5 테스트 셋으로 시스템 평가하기

1. 최고 성능 모델 확인

```
final_model = grid_search.best_estimator_
```

1. 테스트 세트 전처리
    - 전처리 파이프라인의 `transform()` 메서드를 직접 활용
    - **주의**: `fit()` 메서드는 전혀 사용하지 않음

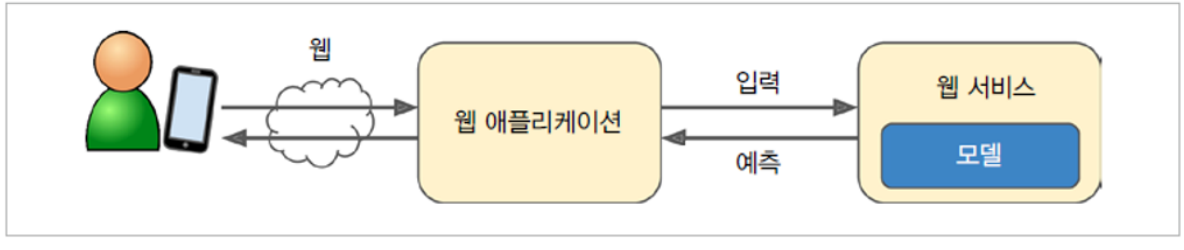
1. 최고 성능 모델을 이용하여 예측하기

1. 최고 성능 모델 평가 및 론칭

최상의 모델 성능 평가

- 테스트 세트에 대한 최고 성능 모델의 RMSE: 47730

최상의 모델 성능 배포



데이터셋 및 모델 백업

- 완성된 모델은 항상 백업해 두어야 함. 업데이트된 모델이 적절하지 않은 경우 이전 모델로 되돌려야 할 수도 있음.
    - 백업된 모델과 새 모델을 쉽게 비교할 수 있음.

- 동일한 이유로 모든 버전의 데이터셋을 백업해 두어야 함.
    - 업데이트 과정에서 데이터셋이 오염될 수 있기 때문임.