핸즈온 머신러닝 1주차 예습 : CH2

2.3.3 데이터 구조 훑어보기

```
# 상위 5행 확인
housing.head()

# 행 수, 각 특성의 데이터 타입과 널이 아닌 값의 개수 확인
housing.info()

# 해당 열(범주형)의 각 카테고리의 개수
housing["ocean_proximity"].value_counts()

# 숫자형 특성의 요약 정보
housing.describe()

# 모든 숫자형 특성에대한 히스토그램 출력 : hist()

%matplotlib inline # 주피터 노트북의 매직 명령
import matplotlib.pyplot as plt
housing.pllt(bins=50, figsize=(20,15))
plt.show()
```

2.3.4 테스트 세트 만들기

: 데이터 스누핑 편향을 방지하기 위해서 무작위로 20%정도의 샘플을 선택하는 것

1. 무작위 샘플링

```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
# random_state는 난수 초기값 지정 위한 매개변수
# 인덱스를 기반으로 데이터셋을 나눌 수 있는 방법
```

2. 계층적 샘플링 stratified sampling

   전체인구 → 계층 strata 라는 동질의 그룹으로 나누고, 각 계층에서 올바른 수의 샘플 추출

- 효과 : 계층별로 데이터셋에 충분한 샘플 수를 획득해 전체인구를 대표하도록
- 주의점 : 너무 많은 계층으로 나누지 말아야 / 각 계층이 충분히 커야

```
housing["income_cat"]=pd.cut(housing["median_income"],
                             bins=[0.,1.5,3.0,4.5,6,np.inf],
                             labels=[1,2,3,4,5])
# income_cat 에 대해 범주를 생성 -> 인덱스 + 해당 범주를 담은 데이터 셋 생성

from sklearn.model_selection import StratifiedShuffleSplit
split=StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index,test_index in split.split(housing, housing["income_cat"]):
    strat_train_set=housing.loc[train_index]
    strat_test_set=housing.loc[test_index]

# 생성된 strat_test_set에는 income_cat 열(지정한 범주) 추가된 상태

strat_test_set["income_cat"].value_counts()/len(strat_test_set)
# 범주별로 전체 테스트 셋에서 차지하는 비율 검토 코드
```

무작위 샘플링 vs 계층적 샘플링 코드

```
def income_cat_proportions(data):
    return data["income_cat"].value_counts() / len(data)

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)

compare_props = pd.DataFrame({
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(strat_test_set),
    "Random": income_cat_proportions(test_set),
}).sort_index()
compare_props["Rand. %error"] = 100 * compare_props["Random"] / compare_props["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] / compare_props["Overall"] - 100
```

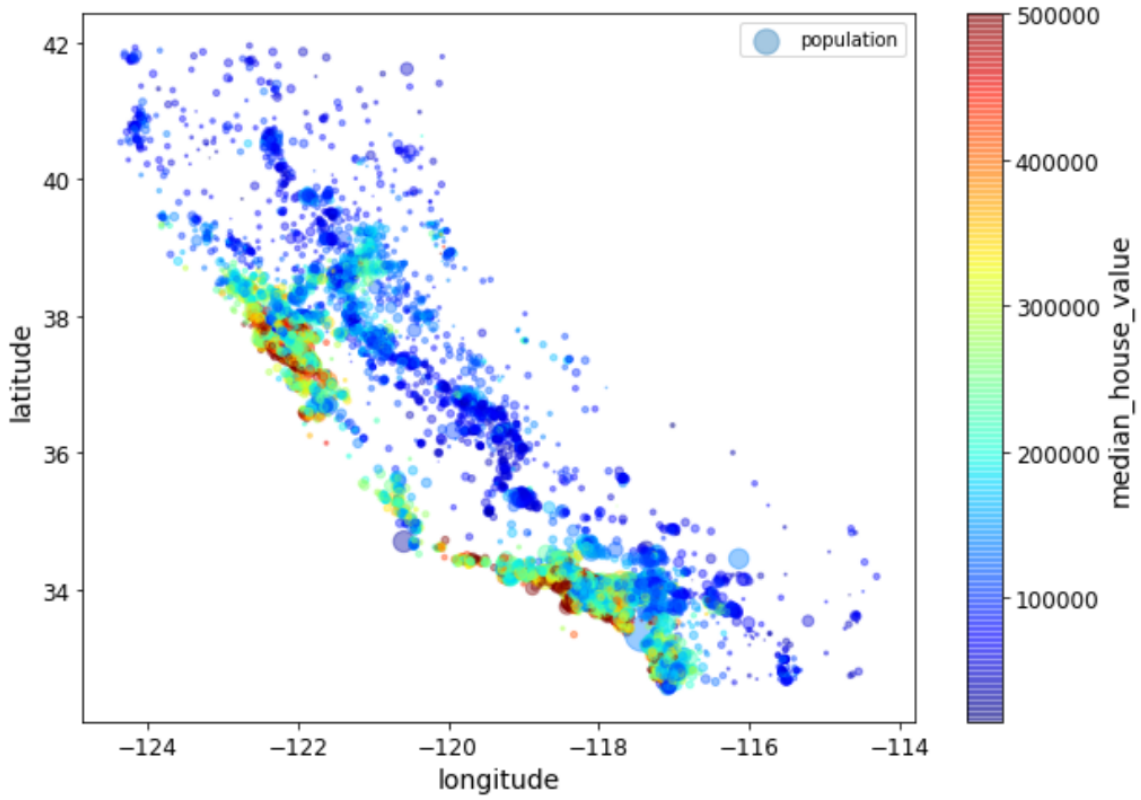
2.4 데이터 시각화

```
housing.plot(kind="scatter",x="longitude",y="latitude",alpha=0.4,

             s=housing["population"]/100, label="population", figsize=(10,7),
              #매개변수: 원의 크기로 popultaion 나타내기

             c="median_house_value",cmap=plt.get_cmap("jet"),colorbar=True,
              #매개변수: 컬러맵 jet로 median_house_value 나타내기

             sharex=False
              # sharex=False 매개변수는 x-축의 값과 범례를 표시하지 못하는 버그를 수정
             )
plt.legend()
```



2.4.2 상관관계 조사

표준 **상관계수** (피어슨의 r) : -1~1

```
corr_matix = housing.corr()

corr_matrix["median_house_value"].sort_values(ascending=False)
#median_house_value 와 다른 특성 사이의 상관관계 큰 순서로 나열
```

- 1에 가까우면 강한 양의 상관관계, -1에 가까우면 강한 음의 상관관계
- 계수가 0에 가까우면 선형적인 상관관계가 없다는 의미
- 상관계수는 선형적인 상관관계만 측정 (비선형 상관관계 잡을 수 X)

```
# 숫자형 특성 사이에 산점도 그리기 : scatter_matrix
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12,8))
```

2.5 머신러닝 알고리즘을 위한 데이터 준비

```
# 머신러닝 알고리즘을 위한 데이터 준비
# 예측변수와 레이블 분리
housing = strat_train_set.drop("median_house_value",axis=1)
# median_house_value 제외 변수들은 예측 변수

housing_labels = strat_train_set["median_house_value"].copy()
#median_house_value는 레이블
```

2.5.1 데이터 정제

누락된 특성 처리

1. 해당 구역 제거
2. 전체 특성 삭제
3. 어떤 값으로 채우기 (0, 평균, 중간값 등)

```
housing.dropna(subset=["total_bedrooms"]) #1

housing.drop("total_bedrooms", axis=1) #2

median = housing["total_bedrooms"].median() #3
housing["total_bedrooms"].fillna(median, inplace=True)
#3 방법의 median 저장하는 것 잊지말기 -> 테스트 세트의 누락값과 실제 데이터 누락값에 필요
```

- sklearn의 **SimpleImputer** 로 누락값 손쉽게 다루기

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
# 누락값을 중간값으로 대체한다고 지정한 상태

housing_num = housing.drop("ocean_proximity", axis=1)
# 중간값이 수치형 특성에서만 계산되기 때문에 텍스트 특성 제외한 복사본 생성
# 다른 방법: housing_num = housing.select_dtypes(include=[np.number])

imputer.fit(housing_num)
# 훈련 데이터에 imputer 객체의 fit() 매서드 적용

imputer.statistics_
# housing_num.median().values와 같은 결과

X = imputer.transform(housing_num)
# imputer 객체를 사용해 훈련세트에서 누락된 값->학습한 중앙값으로 바꿔
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
# 판다스 데이터 프레임으로 바꾸기
```

2.5.2 텍스트와 범주형 특성 다루기

- sklearn의 **Ordinal Encoder**

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder=OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
# 카테고리를 텍스트에서 숫자로 변환

housing_cat_encoded[:10]

ordinal_encoder.categories_
#categories_ 로 카테고리 목록 얻기
```

→ 단점: 머신러닝 알고리즘이 가까이 있는 두 값이 떨여져 있는 두 값보다 더 비슷하다고 생각

→ 해결: one-hot-encoding

- sklearn의 **OneHotEncoder**

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder=OneHotEncoder()
housing_cat_1hot=cat_encoder.fit_transform(housing_cat)
housing_cat_1hot #희소행렬

housing_cat_1hot.toarray() #희소행렬을 넘파이배열로 변환

cat_encoder.categories_
#categories_ 로 카테고리 목록 얻기
```

2.5.4 특성 스케일링

- min_max 스케일링 : **MinMaxScaler**
- 표준화 : **StandardScaler**

주의
모든 변환기에서 스케일링은,
전체 데이터가 아니고 **훈련데이터에 대해서만 fit( )매서드** 적용해야
그런 다음 **훈련 세트와 테스트 세트 (그리고 새로운 데이터)에 대해 transform( ) 매서드** 사용

2.5.5 변환 파이프라인

Pipeline

연속된 변환을 순서대로 처리할 수 있도록 도와주는 클래스

- 숫자형 특성 처리 파이프라인

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

- 범주형+숫자형 처리 파이프라인 : ColumnTransformer

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)       # 수치형 열 이름의 리스트 생성
cat_attribs = ["ocearn_proximity"]    # 범주형 열 이름의 리스트 생성

full_pipeline = ColumnTransformer([
          ("num", num_pipeline, num_attribs),     # 밀집 행렬 반환
          ("cat", OneHotEncoder(), cat_attribs),  # 최소 행렬 반환
      )]                                          # 최종 행렬의 밀집정도 추정 후
                                                    (0이 아닌 원소 비율)
                                                    밀집도<임계값->희소행렬반환
housing_prepared = full_pipeline.fit_transform(housing)
```

2.6 모델 선택과 훈련

- 선형 회귀 모델 훈련

```
from sklearn.linear_model import LinearRegression

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

```
# 훈련 샘플 몇 개를 사용해 전체 파이프라인을 적용해 보겠습니다
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)

print("예측:", lin_reg.predict(some_data_prepared))
print("레이블:", list(some_labels))
```

- 회귀 모델의 RMSE 측정

```
from sklearn.metrics import mean_squared_error
housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error (housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse # 6828.xx -> 매우 큰 값
```

→ RMSE가 매우 큰 경우 : 훈련 데이터에 과소적합된 사례

→ 모델이 강력하지 못함 의미

→ Sol: 더 강력한 모델 선택 / 더 좋은 특성 주입 / 모델 규제 감소

- DecisionTreeRegressor

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error (housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse #0
```

→ 오차가 0 → 과대적합

2.6.2 교차 검증을 사용한 평가

- sklearn의 **k-fold cross-validation**

  : 훈련세트를 10개의 fold(subset)로 무작위 분할 → 결정 트리 모델 10번 훈련+평가

  : 매번 다른 폴드를 선택해 평가에 사용, 나머지 9개 폴드는 훈련에 사용

```
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)
```


사이킷런의 교차 검증은 매개변수가 클수록 좋은 효용 함수 기대
→ MSE의 반댓값인 neg_mean_squared_error 함수 사용

- **RandomForestRegressor**

  : 특성을 무작위로 선택해 많은 결정트리를 만들고 그 예측을 평균 내는 방식으로 작동

```
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)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
```

```
forest_rmse

from sklearn.model_selection import cross_val_score

forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,
                                scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

2.7 모델 세부 튜닝

가능한 모델 추린 후 모델 세부 튜닝

2.7.1 그리드 탐색

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    # 12(=3×4)개의 하이퍼파라미터 조합을 시도합니다.
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # bootstrap은 False로 하고 6(=2×3)개의 조합을 시도합니다.
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor(random_state=42)
# 다섯 개의 폴드로 훈련하면 총 (12+6)*5=90번의 훈련이 일어납니다.
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)

# 최상의 파라미터 조합 출력
grid_search.best_params_

# 최적의 추정기 접근
grid_search.best_estimator_

#평가 점수 확인
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
#-> RMSE 값이 가장 작은 것이 최적의 모델
```

2.7.2 랜덤 선택

- RandomizedSearchCV

  장점: - 랜덤 탐색 반복 횟수만큼 하이퍼파라미터가 값 탐색

  (그리드 탐색은 하이퍼 파라미터마다 몇 개만 탐색)

  - 반복 횟수 조절로 하이퍼파라미터 탐색에 투입할 컴퓨터 자원 제어

2.7.3 앙상블 방법

최상의 모델 연결 (7장)

2.7.4 최상의 모델과 오차 분석

```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # 예전 방식
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)
```

→ 위 정보를 바탕으로 덜 중요한 특성 제외 가능

2.7.5 테스트 세트로 시스템 평가하기

테스트 셋에서 최종 모델 평가

```
final_model = grid_search.best_estimator_

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

X_test_prepared = full_pipeline.transform(X_test) #fit_transform 이 아님 주의
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
final_rmse
```

신뢰구간 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)))
```