[멘멘 2강] 머신러닝 프로젝트 톺아보기 : 아파트 실거래가 예측

다양한 데이터와 전처리 과정을 살펴보기 위해 서울, 부산지역 아파트 빅데이터를 이용합니다.

아파트와 관련된 다양한 데이터들을 종합하여 아파트 실거래가를 예측하는 머신러닝 모델을 만드는 것을 목표로 합니다.

모델보다는 데이터 전처리와 성능 평가 등에 비중을 두었습니다.

1. 데이터 살펴보기

전체 데이터 확인

- head()로 데이터프레임 확인

```
train.head()
```

	transaction_id	apartment_id	city	dong	jibun	apt	addr_kr	exclusive_use_area	year_of_completion	transaction_year_month
0	0	7622	서울특별시	신교동	6-13	신현(101동)	신교동 6-13 신현(101동)	84.82	2002	200801
1	1	5399	서울특별시	필운동	142	사직파크맨션	필운동 142 사직파크맨션	99.17	1973	200801
2	2	3578	서울특별시	필운동	174-1	두레엘리시안	필운동 174-1 두레엘리시안	84.74	2007	200801
3	3	10957	서울특별시	내수동	95	파크팰리스	내수동 95 파크팰리스	146.39	2003	200801
4	4	10639	서울특별시	내수동	110-15	킹스매너	내수동 110-15 킹스매너	194.43	2004	200801

- info()로 변수, 전체 행 수, **데이터 타입**, NULL 값 개수 확인

```
[6]  train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1216553 entries, 0 to 1216552
Data columns (total 13 columns):
 #   Column                  Non-Null Count    Dtype
---  ------                  --------------    -----
 0   transaction_id          1216553 non-null  int64
 1   apartment_id            1216553 non-null  int64
 2   city                    1216553 non-null  object
 3   dong                    1216553 non-null  object
 4   jibun                   1216553 non-null  object
 5   apt                     1216553 non-null  object
 6   addr_kr                 1216553 non-null  object
 7   exclusive_use_area      1216553 non-null  float64
 8   year_of_completion      1216553 non-null  int64
 9   transaction_year_month  1216553 non-null  int64
 10  transaction_date        1216553 non-null  object
 11  floor                   1216553 non-null  int64
 12  transaction_real_price  1216553 non-null  int64
dtypes: float64(1), int64(6), object(6)
memory usage: 120.7+ MB
```

▼ 변수 설명

transaction_real_price : 아파트 실거래가

exclusive_use_area : 전용면적

year_of_completion : 완공된 해

transaction_year_month : 거래년월

transaction_date : 거래일

floor : 층

city : 도시

- **object(범주형)** : value_counts()로 카테고리별 값 확인

```
[7]  train.city.value_counts() #범주형변수
```

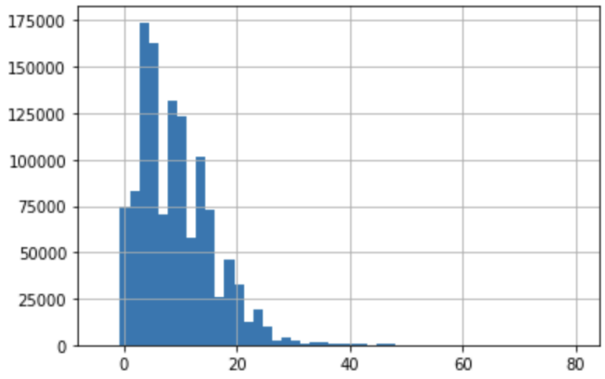
```
서울특별시    742285
부산광역시    474268
Name: city, dtype: int64
```

- **int, float(숫자형)** : describe()로 요약 정보 확인, hist()로 분포를 확인.

```
train.floor.describe() #숫자형
```

```
count    1.216553e+06
mean     9.343291e+00
std      6.606500e+00
min     -4.000000e+00
25%      4.000000e+00
50%      8.000000e+00
75%      1.300000e+01
max      8.000000e+01
Name: floor, dtype: float64
```

```
[9] train.floor.hist(bins=50)
    plt.show() #히스토그램으로 확인
```



- 여기서 사용한 데이터셋은 테스트 세트와 훈련 세트가 이미 나누어져 있는 데이터지만, 훈련 세트를 이용해 훈련시킬 때 자체적으로 검증하기 위해서 다시 검증세트와 훈련세트로 나눠줘야 함. → 샘플링이 필요!

샘플링

1. 무작위 샘플링

- train_test_split(data, test_size, random_state) 를 사용.

```
[14] train_set, test_set = train_test_split(train, test_size=0.2, random_state=42)
     print(train_set.shape)
     print(test_set.shape)

     (973242, 13)
     (243311, 13)
```

- 데이터를 완전 무작위로 샘플링한 것. 데이터의 특성에 따라서는 무작위가 아닌 계층적 샘플링이 필요한 경우가 존재함.

2. 계층적 샘플링

- 서울시와 부산시 아파트 간 유의미한 가격차이가 난다고 가정해보자.

  ▼ 이 데이터에서는 계층적 샘플링과 무작위 샘플링이 큰 차이가 없음.

  - city가 서울 또는 부산 둘 중 하나의 값만 가지기 때문에 무작위 샘플링을 하나, 계층적 샘플링을 하나 비율은 거의 같게 나옴.
  - 다양한 범주를 갖는 변수로 계층을 나눈다면, 계층적 샘플링이 더 효과적일 것.

- 전체 데이터에서 서울지역과 부산지역의 비율에 맞춰 샘플링을 진행할 필요가 있음.
- value_counts()로 훈련 세트 내에서 서울과 부산지역의 데이터 수에서 유의미한 차이가 있다는 것을 확인.

```
[15] train.city.value_counts()

     서울특별시    742285
     부산광역시    474268
     Name: city, dtype: int64
```

- 단, 이때는 서울특별시와 부산광역시가 숫자가 아닌 텍스트이기 때문에 이를 라벨인코딩해서 숫자로 바꿔줘야 함. (인코딩은 뒤에서 자세히 설명)
- StratifiedShuffleSplit() 사용하면 비율에 맞춰 계층적 샘플링이 가능함.

```
split = StratifiedShuffleSplit(n_splits=1, test_size=.2, random_state=10)
for train_idx, test_idx in split.split(train, train['city']):
  strat_train_set = train.loc[train_idx]
  strat_test_set = train.loc[test_idx]
```

- 계층적 샘플링과 무작위 샘플링을 각각 비교해보면, 이 데이터에서는 거의 비슷하게 나옴.

```
#계층적 샘플링 했을때 서울, 부산 각각의 데이터 개수
strat_train_set['city'].value_counts()
```

```
1.0    593828
0.0    379414
Name: city, dtype: int64
```

```
[28] #무작위 샘플링
train_set['city'].value_counts()
```

```
1.0    593666
0.0    379576
Name: city, dtype: int64
```

상관관계 확인

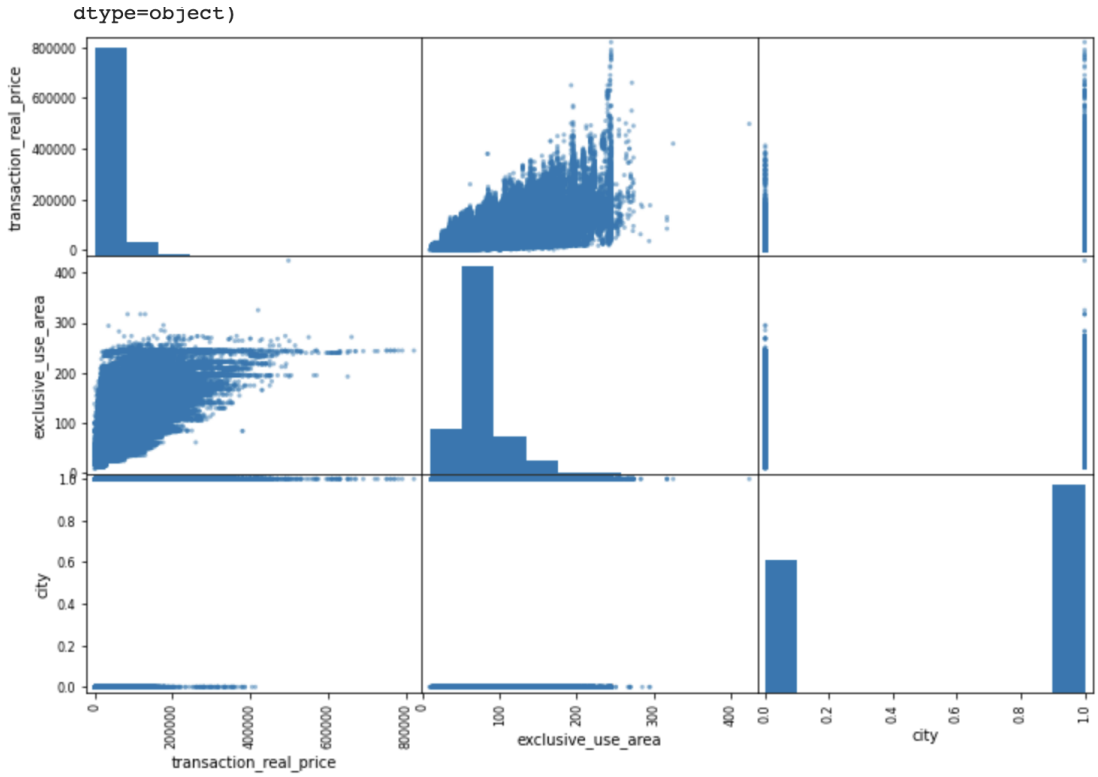
- corr() 사용해 상관계수를 확인, 숫자의 절댓값이 1에 가까울수록 두 변수 간 선형 상관관계가 강함을 의미.

```
#상관계수 확인 - corr() 숫자 확인

corr_matrix = train.corr()
corr_matrix['transaction_real_price'].sort_values(ascending=False)
```

```
transaction_real_price    1.000000
exclusive_use_area        0.561563
city                      0.440886
transaction_year_month    0.183910
floor                     0.112278
year_of_completion        0.052656
apartment_id              0.017576
transaction_id           -0.120734
Name: transaction_real_price, dtype: float64
```

- 위에서 확인한 상관계수 값이 큰 변수들을 scatter_matrix 사용해 시각화

```
#상관계수 확인 - scatter_matrix 시각화

attributes = ['transaction_real_price', 'exclusive_use_area', 'city']
scatter_matrix(train[attributes], figsize=(12,8))
```



- 이렇게 확인한 값들 중 유의미한 관계가 나타나는 변수들을 따로 확인.
  - 전용면적이 클수록 실거래가가 비싸지는 경향.
  - 전반적으로 city==0 보다 city==1일때의 실거래가가 더 높음. (부산보다 서울의 집값이 비싼 경향)
- scatter matrix를 통해 시각화했을때 이상치가 나타나면 제거.(수평선 등)

2. 데이터 전처리

> 🤔 새로운 특성을 만들고, 모델이 학습하기 용이하게 데이터를 전처리 해보자.

아파트 이름(apt)

✅ 같은 아파트 간 다른 동은 가격 차이가 유의미하게 나지 않을 것이라 예상.

- 예를 들면, 신현(101동) 과 신현(102동) 간의 실거래가 차이가 크지 않을 것이라 예상.
- 즉 괄호 내의 세부 동 수는 제거하고, 이렇게 제거한 후 같은 아파트 이름을 갖는 데이터가 얼마나 있는지 확인해보자!

```
train['apt'].value_counts()[:20]
```

```
현대              13329
한신               9878
삼성               6729
대우               6216
신동아              5851
두산               5801
주공2              5669
삼성래미안             5483
우성               5411
벽산               4651
동원로얄듀크             4430
경남               4028
삼환               3896
대림               3800
쌍용               3409
롯데캐슬              3381
삼익               3362
오륙도에스케이뷰            3220
코오롱              3174
파크리오              3094
Name: apt, dtype: int64
```

- 상위 20개의 아파트명만 출력했을때. 아파트 이름이 많이 겹친다는 것을 알 수 있다.

  → **<u>같은 이름을 갖는 아파트가 몇 개나 있는지</u>**를 나타내는 변수를 생성하자.

  - 같은 아파트 이름을 갖는 새로운 feature를 생성

```
#같은 아파트 이름을 갖는 수로 새로운 피쳐 생성

train['apt_counts'] = 0

train.groupby('apt')['apt_counts'].count()
train = pd.merge(train, train.groupby('apt')['apt_counts'].count(),
                 on='apt', how='left').drop('apt_counts_x',axis=1).rename(columns={'apt_counts_y':'apt_counts'})

test['apt_counts'] = 0

test.groupby('apt')['apt_counts'].count()
test = pd.merge(test, test.groupby('apt')['apt_counts'].count(),
                 on='apt', how='left').drop('apt_counts_x',axis=1).rename(columns={'apt_counts_y':'apt_counts'})
```

```
[46] train.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 1216553 entries, 0 to 1216552
Data columns (total 14 columns):
 #   Column                  Non-Null Count    Dtype  
---  ------                  --------------    -----  
 0   transaction_id          1216553 non-null  int64  
 1   apartment_id            1216553 non-null  int64  
 2   city                    1216553 non-null  float64
 3   dong                    1216553 non-null  object 
 4   jibun                   1216553 non-null  object 
 5   apt                     1216553 non-null  object 
 6   addr_kr                 1216553 non-null  object 
 7   exclusive_use_area      1216553 non-null  float64
 8   year_of_completion      1216553 non-null  int64  
 9   transaction_year_month  1216553 non-null  int64  
 10  transaction_date        1216553 non-null  object 
 11  floor                   1216553 non-null  int64  
 12  transaction_real_price  1216553 non-null  int64  
 13  apt_counts              1216553 non-null  int64  
dtypes: float64(2), int64(7), object(5)
memory usage: 139.2+ MB
```

- 같은 이름을 가진 아파트가 많지 않은 경우에는, 이 변수가 의미 없을 가능성이 높다.
  - 상위 50개 아파트명을 제외하고, **나머지는 others로 통일**시킴

    ▼ 상위 50개를 제외한 나머지를 others로 통일시키는 코드

```
apt_name = [ name for name in apt_frame.index]
apt_name #상위 50개 아파트명

#아파트 이름이 apt_name에 들어가지 않으면, 아파트 명을 others로 변경
train['transformed'] = False
test['transformed'] = False

for a in tqdm(apt_name):
  train.loc[train['apt'].str.contains(a),'apt'] = a
  test.loc[test['apt'].str.contains(a), 'apt'] = a
  train.loc[train['apt'].str.contains(a), 'transformed'] = True
  test.loc[test['apt'].str.contains(a), 'transformed'] = True

for a in tqdm(apt_name):
  train.loc[~train['transformed'], 'apt'] = 'others'
  test.loc[~test['transformed'],'apt'] = 'others'
```

  - 이렇게 통일시키고 나서 다시 데이터를 확인하면, 아래와 같이 others로 통일된 것을 볼 수 있음.

```
[51] #다시 데이터 확인
     train['apt'].value_counts()
```

```
others        735228
현대             85530
삼성             33816
한신             27481
동아             27410
벽산             26968
롯데캐슬          25245
우성             24524
대림             23029
두산             20691
대우             19986
삼익             18708
쌍용             18256
경남             10875
극동              9925
주공2             9912
한양              9376
코오롱             8765
청구              8618
성원              7697
삼환              6073
동원로얄듀크          6070
```

- 아파트명이 같은 아파트들의 '평균 가격'을 계산하고, 평균 가격이 높은 순서대로 아파트명을 라벨인코딩.

```
[52] apt_price = train.groupby('apt')['transaction_real_price'].agg('mean').sort_values(ascending=False)
     print('변환전\n', apt_price[:5])

     for i, a in enumerate(list(apt_price.index)):
       train.loc[train['apt']==a, 'apt'] = i #라벨인코딩
       test.loc[test['apt'] ==a, 'apt'] = i

     apt_price = train.groupby('apt')['transaction_real_price'].agg('mean').sort_values(ascending=False)
     print('변환후\n', apt_price[:5])
```

```
변환전
 apt
잠실엘스         96498.002657
리센츠          92961.140555
파크리오         88739.646736
개포주공 1단지     88516.395161
더샵센텀파크1차     53045.140594
Name: transaction_real_price, dtype: float64
변환후
 apt
0    96498.002657
1    92961.140555
2    88739.646736
3    88516.395161
4    53045.140594
Name: transaction_real_price, dtype: float64
```

  - '아파트의 평균 실거래가'라는 순서가 있기 때문에 원핫벡터가 아니라 라벨인코딩
  - 가장 평균 가격이 높은 잠실엘스를 시작으로 **아파트명**을 0,1,2,..의 **정수로 변환**

날짜 관련 변수(date)

✅ 완공년도, 거래년월, 거래일 등의 날짜 관련 변수도 인코딩 진행

year_of_completion	transaction_year_month	transaction_date
2002	200801	21~31
1973	200801	1~10

- 각각의 최대연도에서 최소연도를 빼면, 정수형으로 라벨인코딩 가능.
- 이렇게 라벨인코딩을 진행하면,
  - 변환전 2002년이 변환후에는 41로,
  - 변환전 2017년 11월이 변환후 118로 바뀜.

```
변환전
 [2002 1973 2007 2003 2004]
변환후
 [41 12 46 42 43]
train 변환전
 [200801 200802 200803 200804 200805]
test 변환전
 [201711 201708 201710 201707 201712]
train 변환후
 [0 1 2 3 4]
test 변환후
 [118 115 117 114 119]
```

동 (dong)

✅ 혹시 서울과 부산에 같은 이름을 가진 동들이 있지 않을까?

- 확인해 보니, ‘송정동', ‘사직동', ‘부암동', ‘중동' 의 4개 동의 이름이 같다.

```
[36] #같은 이름을 가진 동이 있는지 확인
     seoul_set = set(train.loc[train['city']=='서울특별시', 'dong'])
     busan_set = set(train.loc[train['city']=='부산광역시', 'dong'])
     same_dong = seoul_set & busan_set
     print(same_dong)

     seoul_set = set(test.loc[test['city']=='서울특별시', 'dong'])
     busan_set = set(test.loc[test['city']=='부산광역시', 'dong'])
     same_dong = seoul_set & busan_set
     print(same_dong)
```

```
{'송정동', '사직동', '부암동', '중동'}
{'송정동', '사직동', '부암동', '중동'}
```

- 이를 구분하기 위해서 앞에 '서울', '부산'을 붙여준다.

```
#위 네 동은 부산과 서울에 이름이 겹침. 앞에 서울특별시, 부산광역시를 붙여서 분리

for d in same_dong:
  train.loc[(train['city']=='서울특별시')&(train['dong']==d),'dong'] = '서울'+d
  train.loc[(train['city']=='부산광역시')&(train['dong']==d),'dong'] = '부산'+d
  test.loc[(test['city']=='서울특별시')&(test['dong']==d),'dong'] = '서울'+d
  test.loc[(test['city']=='부산광역시')&(test['dong']==d),'dong'] = '부산'+d
```

- 동별로 아파트 실거래가의 평균을 확인해 보니, 동별로 가격차이가 꽤 많이 나는 것을 볼 수 있음.
  → *어느 동인지도 아파트 가격에 크게 영향을 미친다고 예측*해볼 수 있음.(당연함..)

```
#동별로 가격차이가 많이 남. -> 동은 그대로 냅두자.. -> 어디가 젤 비쌈?

dong_price = train.groupby('dong')['transaction_real_price'].agg('mean').sort_values(ascending=False)
dong_price[:20]
```

```
dong
장충동1가     269888.888889
압구정동      164534.722914
청암동       161403.700000
용산동5가     153497.331633
회현동2가     139906.140351
반포동       132489.395651
한남동       122593.293264
서빙고동      116547.239777
대치동       116320.538909
남대문로5가    113153.604651
도곡동       110655.655354
청담동       110289.411168
교남동       108600.000000
내수동       107899.014778
주성동       106000.000000
하중동       105900.477833
삼성동       101379.971621
잠실동       101166.337275
동자동        99472.876033
동빙고동       99057.552239
Name: transaction_real_price, dtype: float64
```

- 동 이름도 텍스트이기 때문에, 이전의 아파트와 마찬가지로 라벨인코딩을 해줌.

```
#라벨 인코딩 진행

for i, d in tqdm(enumerate(list(dong_price.index)), total=len(dong_price)):
  train.loc[train['dong']==d, 'dong']=i
  test.loc[test['dong']==d, 'dong']=i
train.head(5)
```

```
100%|██████████| 477/477 [00:43<00:00, 10.86it/s]
```

	transaction_id	apartment_id	city	dong	apt	exclusive_use_area	year_of_
0	0	7622	서울특별시	138	17	84.82	
1	1	5399	서울특별시	65	17	99.17	
2	2	3578	서울특별시	65	17	84.74	
3	3	10957	서울특별시	13	17	146.39	

- dong이 다 숫자로 바뀐것을 볼 수 있음.

층 (floor)

✅ 데이터를 살펴보면, floor(층) 값이 음수인 경우가 존재함. (-4층?)

```
train.floor.describe()
```

```
count    1.216553e+06
mean     9.343291e+00
std      6.606500e+00
min     -4.000000e+00
25%      4.000000e+00
50%      8.000000e+00
75%      1.300000e+01
max      8.000000e+01
Name: floor, dtype: float64
```

- 음수를 양수로 바꿔주기 위해서 모든 데이터에 같은 숫자를 더해주고, 라벨 인코딩 (+4)

```
print('변환전\n', train['floor'].values[:5])
train['floor'] = train['floor'].map(lambda x: x+4)
test['floor'] = test['floor'].map(lambda x: x+1)
print('변환후\n', train['floor'].values[:5])
```

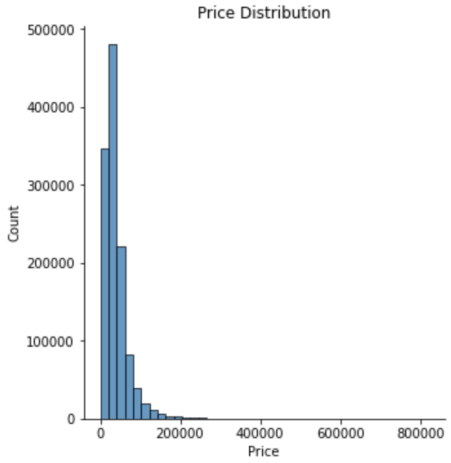
```
변환전
 [ 2  6  6 15  3]
변환후
 [ 6 10 10 19  7]
```

가격 (Price)

- seaborn으로 시각화해서 가격대별로 데이터가 얼마나 존재하는지 확인

```
plt.figure()
sns.displot(train['transaction_real_price'], bins=40)
plt.xlabel('Price')
plt.title('Price Distribution')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```

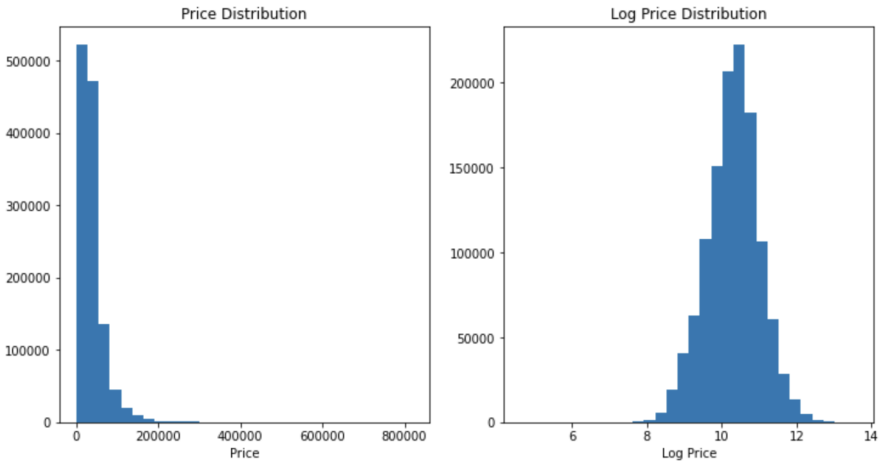


✅ 한쪽에 치우친 모양을 보임.

▼ **정규화**

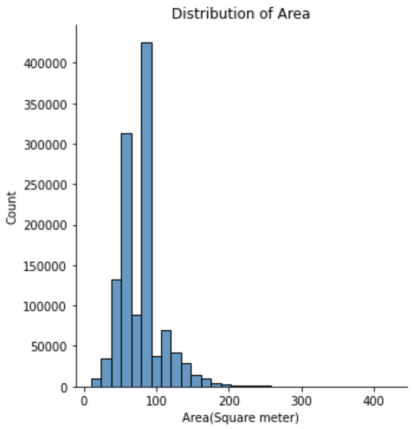
- min-max scaling : 0과 1 사이에 오도록 MinMaxScaler 를 활용해 정규화
- Standardization : 평균이 0, 분산이 1이 되도록 StandardScaler를 활용해 표준화
- 이외에도 다양한 정규화 방법이 존재함.
- 이 데이터처럼 한쪽으로 치우친 값들은 로그변환을 통해 적절한 분포로 만들어주는 과정이 필요!

- 왼쪽은 가격, 오른쪽은 로그변환시킨 가격. 로그변환 후 정규분포에 가까운 모양이 나타남.



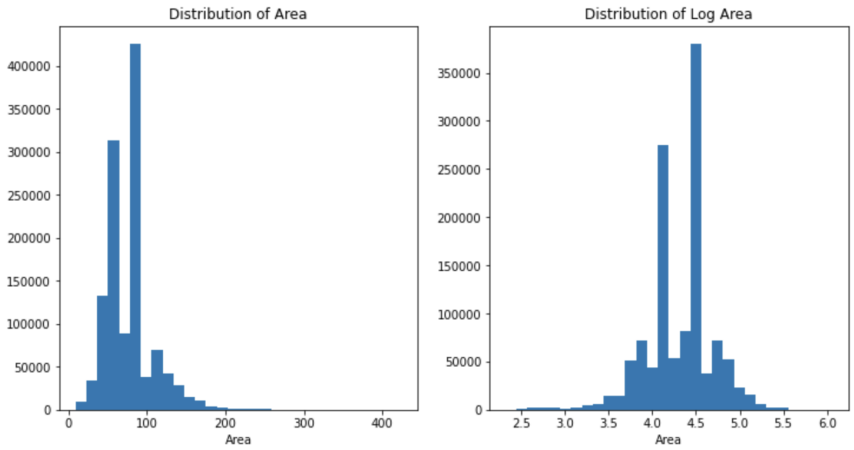
면적 (Area)

- 위와 마찬가지로, 시각화해서 면적별 데이터가 얼마나 존재하는지 확인



✔️ 한쪽에 치우친 모양을 보임.

- 마찬가지로 로그변환하고 분포 다시 확인해 보면, 이전보다 덜 치우침.



도시 (city)

✔️ 도시 이름은 서울특별시 or 부산광역시 둘 중 하나

- 서울특별시면 1, 부산광역시면 0으로 변경

```
train['city']=train['city'].map(lambda x:1 if x=='서울특별시' else 0)
test['city'] = test['city'].map(lambda x:1 if x=='서울특별시' else 0)
```

	city	dong	apt	year_of_completion	transaction_year_month	floor	log_price	log_area
0	1	138	17	41	0	6	10.532123	4.452252
1	1	65	17	12	0	10	9.903538	4.606869
2	1	65	17	46	0	10	10.558439	4.451319
3	1	13	17	42	0	19	11.678448	4.993082
4	1	13	17	43	0	7	11.695255	5.275202

- 텍스트 특성을 인코딩할때는 각각의 순서가 유의미한지를 확인 후 인코딩 결정.
  - 라벨인코딩 : 순서도 의미있는 경우 (중졸,고졸,대졸,박사 등)
    - 중졸 = 0, 고졸 = 1, 대졸 = 2, 박사 = 3 등으로 인코딩
  - 원핫인코딩 : 순서가 정해져 있지 않은 경우(서울시,부산시,성남시 등)
    - 서울시 = [1,0,0], 부산시 = [0,1,0], 성남시 = [0,0,1] 등으로 인코딩
- 이 데이터는 서울, 부산 간 순서가 정해져 있지 않아 원핫 인코딩을 해도 되지만, 값이 두 개 뿐이므로 그냥 라벨 인코딩해도 무관함.

전처리 과정을 마친 후

- 필요없는 컬럼들은 drop()으로 삭제

```
drop_col = ['transaction_id', 'apartment_id', 'apt_counts','transformed']
train.drop(drop_col, axis=1, inplace=True)
test.drop(drop_col, axis=1, inplace=True)
train.head(5)
```

	city	dong	apt	year_of_completion	transaction_year_month	floor	log_price	log_area
0	1	138	17	41	0	6	10.532123	4.452252
1	1	65	17	12	0	10	9.903538	4.606869
2	1	65	17	46	0	10	10.558439	4.451319
3	1	13	17	42	0	19	11.678448	4.993082
4	1	13	17	43	0	7	11.695255	5.275202

- 마지막에 train, test set의 shape 확인

```
[52] print(train.shape, test.shape)

     (1216553, 8) (5463, 7)
```

  - train set : label이 되는 '아파트 실거래가'를 포함해 8개 컬럼 존재
  - test set : label을 제외한 7개 컬럼 존재

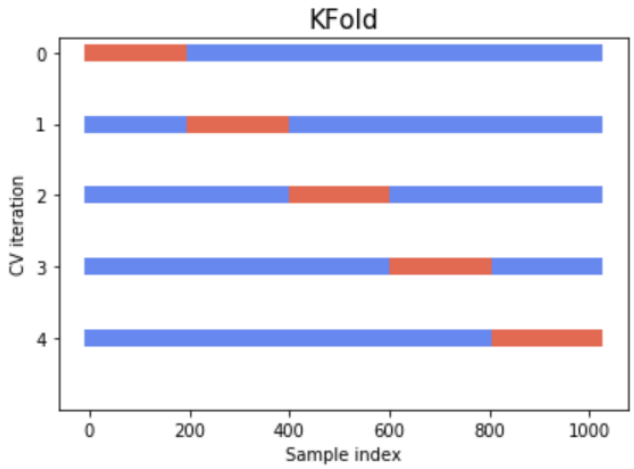
3. 모델 선택 및 훈련, 평가

모델 훈련

- Linear Regression
- Decision Tree
- Random Forest
- 위 세 가지 모델을 사용해 훈련하고, 평가해보자.

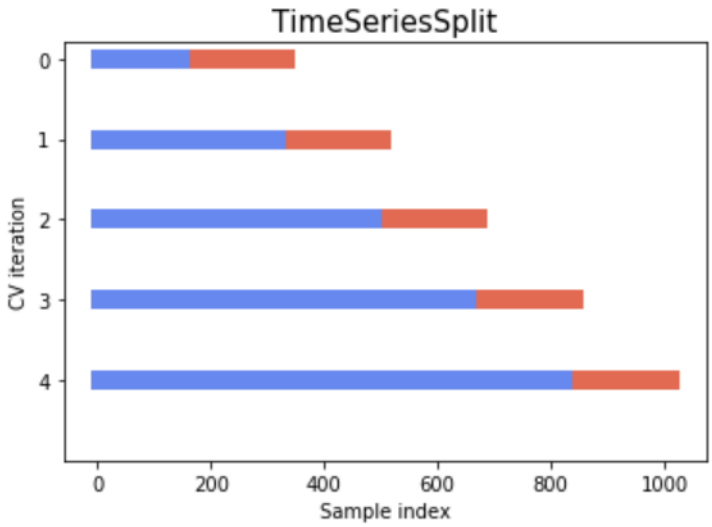
교차 검증을 사용한 평가

- 모델의 평가 → 서로 다른 모델을 사용해서 결과를 낼 때, 모델 중 어떤게 더 나은지 알기 위해서 평가해야함
- 이때 사용되는 평가방법이 교차검증.
- **k-fold-cross-validation**



  - 훈련 세트를 n개의 subset으로 무작위 분할
  - 결정 트리 모델을 n번 훈련하고 평가
  - 매번 다른 폴드를 선택해서 평가에 사용하고, 나머지 n-1개를 훈련에 사용.
  - 이렇게 해서 총 n번 하면, n개의 평가 점수가 담긴 결과가 나옴.
- 교차 검증을 사용하면, 모델의 성능을 추정할 수 있으며 이 추정이 얼마나 정확한지 그 표준 편차도 측정 가능.

- **TimeSeries Cross validation**



- 시계열 데이터에서 일반적인 k-fold를 사용하게 되면, 미래 가격으로 과거를 예측하는 경우가 발생.
- 이렇게 모델을 학습시키면 모델의 정확도는 높아질 지 몰라도, 실제로 사용할 수 없음.
- 따라서 시계열 데이터는 TimeSeriesSplit을 사용해 교차검증해야 함

모델 세부 튜닝

- 가능성 있는 모델들을 가지고 다시 세부 튜닝하는 과정이 필요.

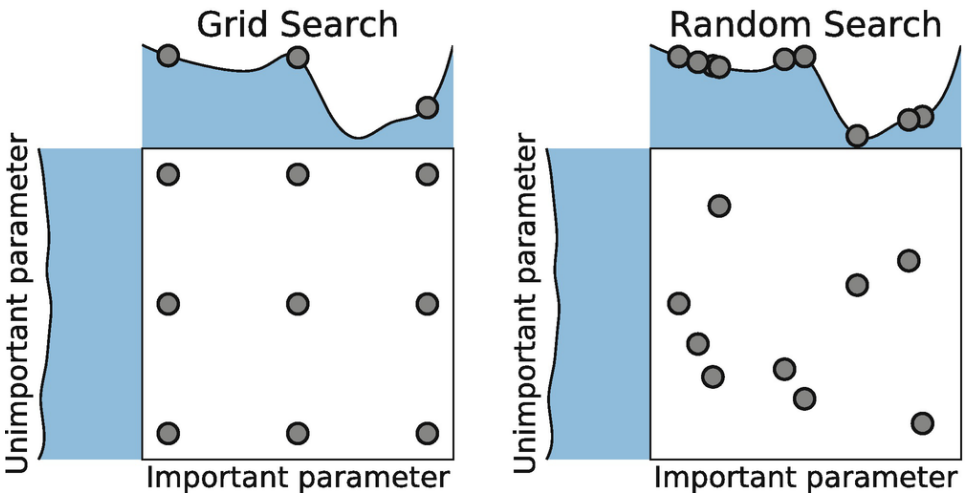
- 예를 들어 결정나무 모델을 선택했다고 가정해보자. 파라미터를 적절하게 선택하면 더 낮은 오차를 얻을 수 있음.
- 적절한 파라미터 값은 어떻게 결정할 수 있을까? → Grid Search, Random Search

1. Grid Search

   적절한 파라미터 값들을 정해주면, 만들 수 있는 모든 파라미터 조합에 대해 교차검증

2. Random Search

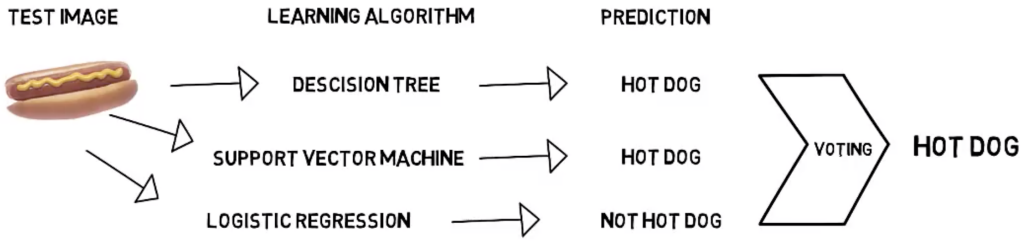
   매번 하이퍼파라미터에 임의의 수를 대입해서 지정 횟수만큼 평가. (값 지정이 필요 없음)



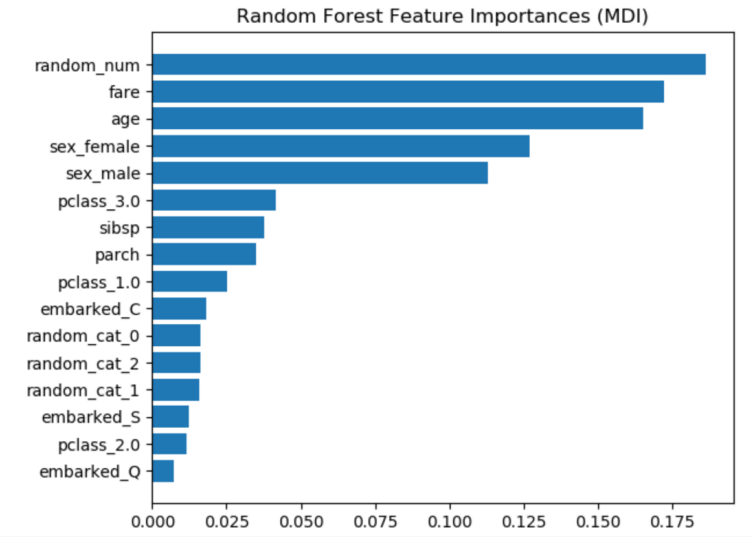
- 이렇게 Important parameter를 찾는 과정을 거쳐, 가장 낮은 오차가 나오는 파라미터 조합을 선택한다.

- 여러 모델의 예측을 합해 최종 예측을 내는 **앙상블 기법**을 활용해 정확도를 더 높일 수 있다.

  (7장에서 더 자세히 다룰 예정)



- feature_importances_를 통해 특성별 중요도를 확인하고, 덜 중요한 특성들은 학습에서 제외시킬 수 있음.



https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance.html

- 예를 들어 위처럼 특성 중요도를 시각화해서 확인한 후, 가장 아래에 있는 특성부터 제거하면서 모델 성능이 어떻게 변하는지 관찰할 수 있음.

성능 지표

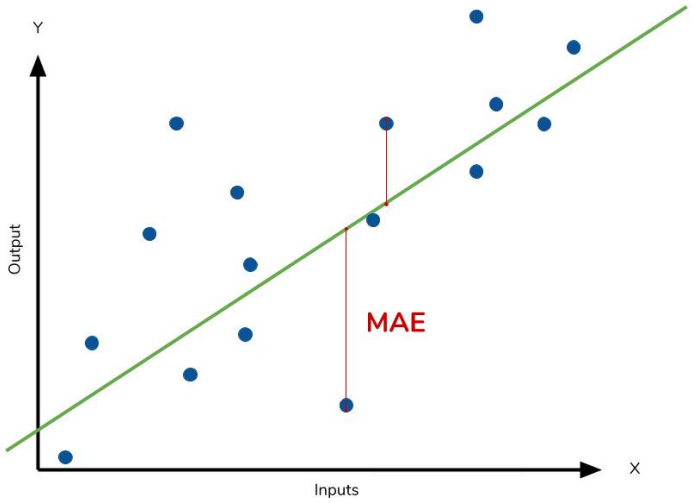
- 모델의 성능을 평가하기 위한 다양한 평가 지표들이 존재한다.

▼ 평가 지표들 보기👀

- MSE 계열 : MSE, MSLE, RMSE, RMSLE
- MAE 계열 : MAE, MAPE
- MASE
- R^2

1. **MAE** (Mean Absolute Error)

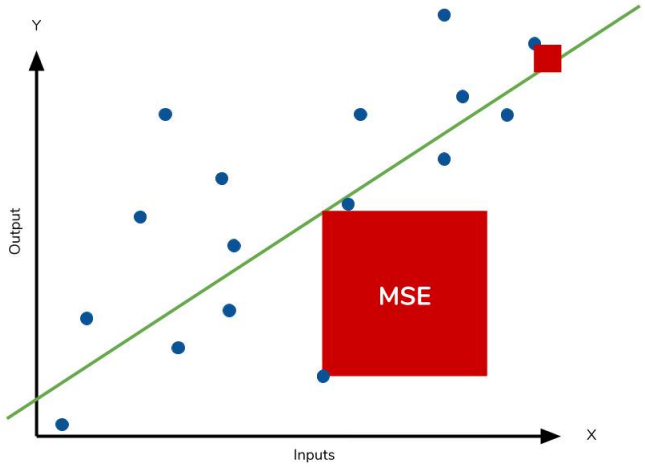
$$MAE = \frac{|(y_i - y_p)|}{n}$$



- 실제 값과 예측값의 차이를 절댓값으로 변환하고, 이를 평균냄.
- 에러에 절댓값을 취하기 때문에 에러의 크기가 그대로 반영됨.
- 따라서 에러가 현실적이고 직관적인 에러일때는 MAE를 사용하는것이 적절함.
- 예를 들어, 아파트의 거래가격을 예측하는 모델에서 MAE가 5백만이라면, ± 5백만 정도의 오차를 예상할 수 있음.
- 그러나 아래의 RMSE를 사용하는 경우, 여기서 계산한 오차를 바로 실생활에 적용할 수 없음.

2. **RMSE** (Root Mean Squared Error)

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\mathrm{Y}i - \hat{\mathrm{Y}}i)^2}$$



(위 그래프에서 그릴 수 있는 모든 빨간 사각형 넓이의 평균에 루트를 씌운 값)

- 이상치 데이터에 모델이 크게 영향을 받으면, 모델의 성능이 저하될 우려가 있음.
- RMSE는 다소 직관성은 떨어지지만,이상치에 대해 더 크게 패널티를 부과한다는 장점이 있음.

→ 예측하고자 하는 데이터의 특성에 맞게 적절한 성능 지표를 선택하는 것이 중요하다✨

▼ 참고자료

핸즈온 머신러닝 2장

https://data101.oopy.io/mae-vs-rmse

https://dailyheumsi.tistory.com/167

https://dacon.io/competitions/official/21265/overview/description

https://medium.com/@soumyachess1496/cross-validation-in-time-series-566ae4981ce4

https://medium.com/@cjl2fv/an-intro-to-hyper-parameter-optimization-using-grid-search-and-random-search-d73b9834ca0a

https://dailyheumsi.tistory.com/111