[파머완] 05 회귀 실습

진규빈 · 방금 전

(9) 회귀 실습 - 자전거 대여 수요 예측

> 캐글 bike-sharing-demand 데이터셋 사용
> => (2011년 1월 ~ 2012년 12월) 날짜/시간, 기온, 습도, 풍속 등의 정보 기반으로 1시간 간격 동안의 자전거 대여 횟수 기재

- datetime: hourly date + timestamp
- season: 1 = 봄, 2 = 여름, 3 = 가을, 4 = 겨울
- holiday: 1 = 토, 일요일의 주말을 제외한 국경일 등의 휴일, 0 = 휴일이 아닌 날
- workingday: 1 = 토, 일요일의 주말 및 휴일이 아닌 주중, 0 = 주말 및 휴일
- weather:
  - 1 = 맑음, 약간 구름 낀 흐림
  - 2 = 안개, 안개 + 흐림
  - 3 = 가벼운 눈, 가벼운 비 + 천둥
  - 4 = 심한 눈/비, 천둥/번개
- temp: 온도(섭씨)
- atemp: 체감온도(섭씨)
- humidity: 상대습도
- windspeed: 풍속
- casual: 사전에 등록되지 않는 사용자가 대여한 횟수
- registered: 사전에 등록된 사용자가 대여한 횟수
- count: 대여 횟수

데이터 클렌징 및 가공과 데이터 시각화

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
warnings.filterwarnings("ignore",category=RuntimeWarning)

bike_df=pd.read_csv(r'bike_train.csv')
print(bike_df.shape)
bike_df.head()
```

(10886, 12)

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

bike_train.csv 업로드 후 DataFrame으로 로드해 데이터 확인

- 10886개 레코드와 12개 칼럼으로 구성

```
bike_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   datetime    10886 non-null  object 
 1   season      10886 non-null  int64  
 2   holiday     10886 non-null  int64  
 3   workingday  10886 non-null  int64  
 4   weather     10886 non-null  int64  
 5   temp        10886 non-null  float64
 6   atemp       10886 non-null  float64
 7   humidity    10886 non-null  int64  
 8   windspeed   10886 non-null  float64
 9   casual      10886 non-null  int64  
 10  registered  10886 non-null  int64  
 11  count       10886 non-null  int64  
dtypes: float64(3), int64(8), object(1)
memory usage: 1020.7+ KB
```

데이터 칼럼 타입 확인

```
# 문자열을 datetime 타입으로 변경
bike_df['datetime']=bike_df.datetime.apply(pd.to_datetime)

# datetime 타입에서 년, 월, 일, 시간 추출
bike_df['year']=bike_df.datetime.apply(lambda x:x.year)
bike_df['month']=bike_df.datetime.apply(lambda x:x.month)
bike_df['day']=bike_df.datetime.apply(lambda x:x.day)
bike_df['hour']=bike_df.datetime.apply(lambda x:x.hour)
bike_df.head(3)
```

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count	year	month	day	hour
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16	2011	1	1	0
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40	2011	1	1	1
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32	2011	1	1	2

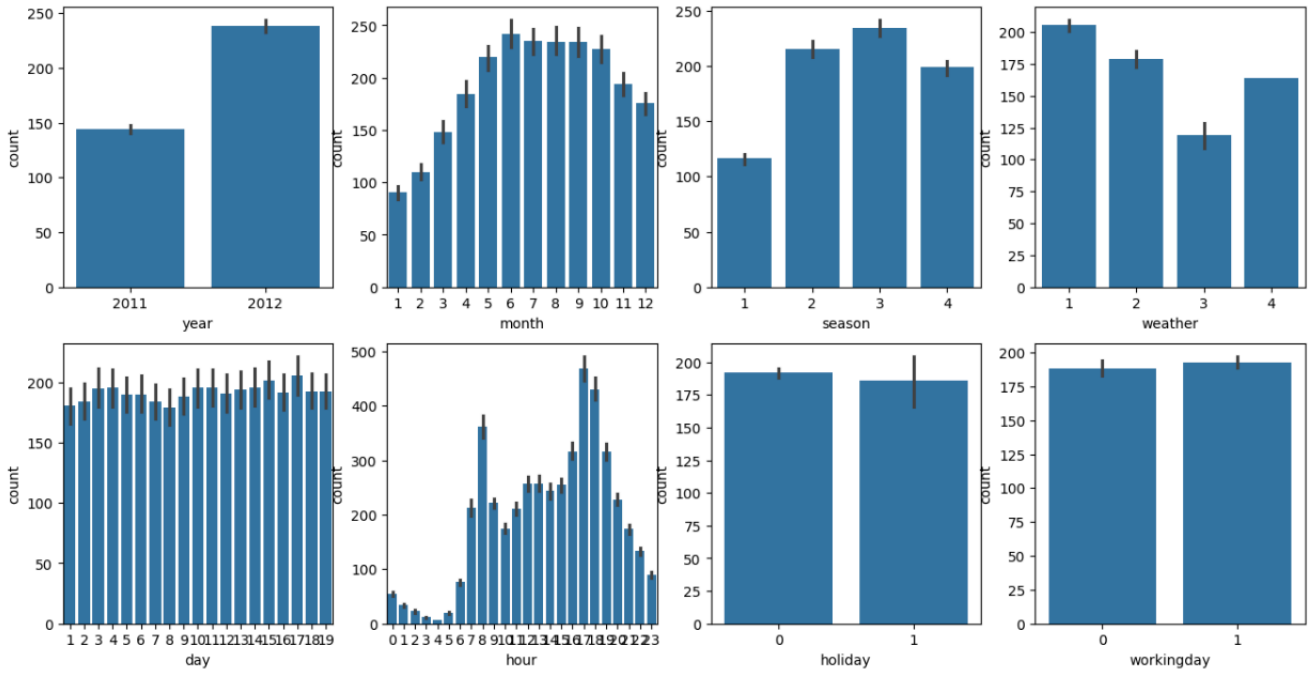
datetime을 년, 월, 일, 시간 4개의 속성으로 분리

- 문자열을 datetime 타입으로 변환하는 판다스 메서드 `apply` 활용

```
drop_columns=['datetime','casual','registered']
bike_df.drop(drop_columns,axis=1,inplace=True)
```

datetime 칼럼 삭제
상관도 높아 예측 저해할 우려 있는 casual & registered 칼럼 삭제

```
fig,axs=plt.subplots(figsize=(16,8),ncols=4,nrows=2)
cat_features=['year','month','season','weather','day','hour','holiday','workingday']

# cat_features에 있는 모든 칼럼별로 개별 칼럼값에 따른 count의 합을 barplot으로 시각화
for i,feature in enumerate(cat_features):
  row=int(i/4)
  col=i%4
  # 시본의 barplot을 이용해 칼럼값에 따른 count의 합을 표현
  sns.barplot(x=feature,y='count',data=bike_df,ax=axs[row][col])
```



주요 칼럼별로 Target 값인 count(대여 횟수) 분포 확인

- 칼럼별 값에 따른 count 값 표현 위해 시본의 barplot 적용
- matplotlib의 `subplots()` 기반으로 시각화

```
from sklearn.metrics import mean_squared_error,mean_absolute_error

# log 값 변환 시 NaN 등의 이슈로 log()가 아닌 log1p()를 이용해 RWSLE 계산
def rmsle(y,pred):
  log_y=np.log1p(y)
  log_pred=np.log1p(pred)
  squared_error=(log_y-log_pred)**2
  rmsle=np.sqrt(np.mean(squared_error))
  return rmsle

# 사이킷런의 mean_square_error()를 이용해 RWSE 계산
def rmse(y,pred):
  return np.sqrt(mean_squared_error(y,pred))

# MAE, RMSE, RMSLE를 모두 계산
def evaluate_regr(y,pred):
  rmsle_val=rmsle(y,pred)
  rmse_val=rmse(y,pred)
  # MAE는 사이킷런의 mean_absolute_error()로 계산
  mae_val=mean_absolute_error(y,pred)
  print('RMSLE:{0:.3f},RMSE:{1:.3F},MAE:{2:.3F}'.format(rmsle_val,rmse_val,mae_val))
```

다양한 회귀 모델 데이터셋에 적용해 예측 성능 측정

- 사이킷런은 RMSLE(Root Mean Square Log Error) 적용하지 않으므로 RMSLE, MAE, RMSE 수행하는 성능 평가 함수 직접 구현

```
# 다음과 같은 rmsle 구현은 오버플로나 언더플로 오류를 발생하기 쉽습니다
def rmsle(y,pred):
  msle=mean_squared_log_error(y,pred)
  rmsle=np.sqrt(mse)
  return rmsle
```

`rmsle()` 구할 때 데이터 값 크기에 따라 오버플로/언더플로 오류 발생하기 쉬운 점 주의
=> `log()` 보다는 `log1p()` 이용

로그 변환, 피처 인코딩과 모델 학습/예측/평가

```
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.linear_model import LinearRegression,Ridge,Lasso

y_target=bike_df['count']
X_features=bike_df.drop(['count'],axis=1,inplace=False)

X_train,X_test,y_train,y_test=train_test_split(X_features,y_target,test_size=0.3,random_state=0)

lr_reg=LinearRegression()
lr_reg.fit(X_train,y_train)
pred=lr_reg.predict(X_test)

evaluate_regr(y_test,pred)
```

```
RMSLE:1.165,RMSE:140.900,MAE:105.924
```

사이킷런 `LinearRegression` 객체 이용해 회귀 예측

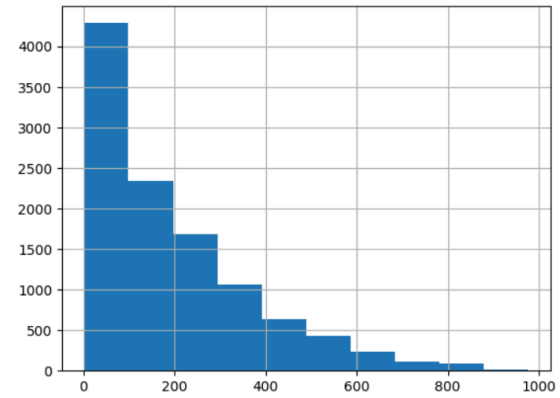
```
def get_top_error_data(y_test,pred,n_tops=5):
  # DataFrame의 칼럼으로 실제 대여 횟수(count)와 예측값을 서로 비교할 수 있도록 생성
  result_df=pd.DataFrame(y_test.values,columns=['real_count'])
  result_df['predicted_count']=np.round(pred)
  result_df['diff']=np.abs(result_df['real_count']-result_df['predicted_count'])

  # 예측값과 실제 값이 가장 큰 데이터 순으로 출력
  print(result_df.sort_values('diff',ascending=False)[:n_tops])

get_top_error_data(y_test,pred,n_tops=5)
```

```
      real_count  predicted_count   diff
1618         890            322.0  568.0
966          884            327.0  557.0
3151         798            241.0  557.0
412          745            194.0  551.0
2817         856            310.0  546.0
```

실제 값과 예측값 어느 정도 차이 나는지 DataFrame의 칼럼으로 만들어 오류 값 가장 큰 순으로 5개만 확인

```
y_target.hist()
```

```
<Axes: >
```

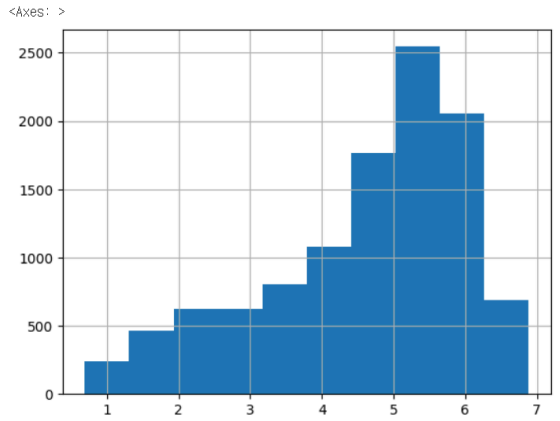


판다스 DataFrame의 `hist()` 이용해 count 칼럼이 정규 분포 이루는지 확인

- count 칼럼 값 0~200 사이에 왜곡 발생 => 넘파이 `log1p()` 이용해 로그 적용 변환 => 변경된 Target 값 기반으로 학습, 예측한 값은 다시 `expm1()` 함수 적용해 원래 scale 값으로 원상 복구

```
y_log_transform=np.log1p(y_target)
y_log_transform.hist()
```

```
<Axes: >
```



`log1p()` 적용한 count 값의 분포 확인

- 변환 전보다 왜곡 정도 많이 향상

```
# 타깃 칼럼인 count 값을 log1p로 로그 변환
y_target_log=np.log1p(y_target)

# 로그 변환된 y_target_log를 반영해 학습/테스트 데이터 세트 분할
X_train,X_test,y_train,y_test=train_test_split(X_features,y_target_log,test_size=0.3,random_state=0)
lr_reg=LinearRegression()
lr_reg.fit(X_train,y_train)
pred=lr_reg.predict(X_test)

# 테스트 데이터 세트의 Target 값은 로그 변환됐으므로 다시 expm1을 이용해 원래 스케일로 변환
y_test_exp=np.expm1(y_test)
# 예측값 역시 로그 변환된 타깃 기반으로 학습돼 예측됐으므로 다시 expm1로 스케일 변환
pred_exp=np.expm1(pred)

evaluate_regr(y_test_exp,pred_exp)
```

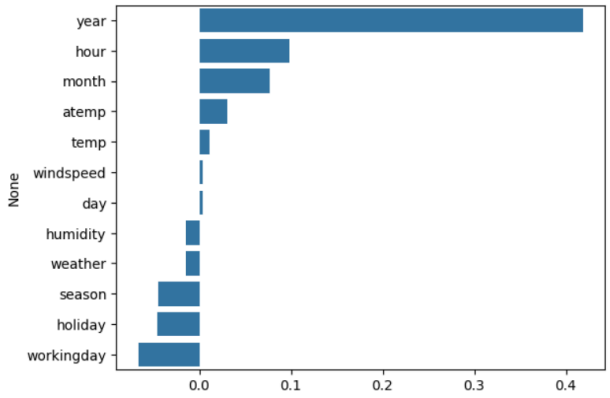
```
RMSLE:1.017,RMSE:162.594,MAE:109.286
```

다시 학습 후 평가 수행

- RMSLE 오류는 줄어들었지만 RMSE는 오히려 더 늘어남

```
coef=pd.Series(lr_reg.coef_,index=X_features.columns)
coef_sort=coef.sort_values(ascending=False)
sns.barplot(x=coef_sort.values,y=coef_sort.index)
```

```
<Axes: ylabel='None'>
```



개별 피처들의 인코딩 적용

- 각 피처의 회귀 계숫값 시각화

```
# 'year','month','day','hour' 등의 피처들을 One Hot Encoding
X_features_ohe=pd.get_dummies(X_features,columns=['year','month','day','hour','holiday','worki
```

판다스 `get_dummies()` 이용해 year, month, day, hour, holiday, workingday, season,weather 칼럼 모두 원-핫 인코딩

- 사이킷런은 카테고리만을 위한 데이터 타입 없으며 모두 숫자로 변환해야 함
- BUT 숫자형 카테고리 값 선형 회귀에 사용할 경우 회귀 계수 연산할 때 이 숫자형 값에 크게 영향 받는 경우 발생 가능

```
# 원-핫 인코딩이 적용된 피처 데이터 세트 기반으로 학습/예측 데이터 분할
X_train,X_test,y_train,y_test=train_test_split(X_features_ohe,y_target_log,test_size=0.3,random_state=0)

# 모델과 학습/테스트 데이터 세트를 입력하면 성능 평가 수치를 반환
def get_model_predict(model,X_train,X_test,y_train,y_test,is_expm1=False):
  model.fit(X_train,y_train)
  pred=model.predict(X_test)
  if is_expm1:
    y_test=np.expm1(y_test)
    pred=np.expm1(pred)
  print('###',model.__class__.__name__,'###')
  evaluate_regr(y_test,pred)
# end of function get_model_predict

# 모델별로 평가 수행
lr_reg=LinearRegression()
ridge_reg=Ridge(alpha=10)
lasso_reg=Lasso(alpha=0.01)

for model in [lr_reg,ridge_reg,lasso_reg]:
  get_model_predict(model,X_train,X_test,y_train,y_test,is_expm1=True)
```

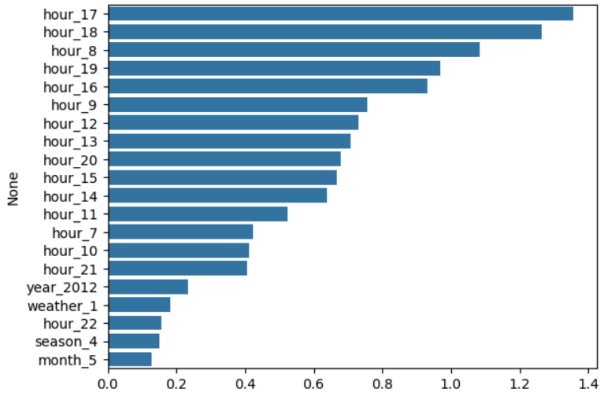
```
### LinearRegression ###
RMSLE:0.590,RMSE:97.688,MAE:63.382
### Ridge ###
RMSLE:0.590,RMSE:98.529,MAE:63.893
### Lasso ###
RMSLE:0.635,RMSE:113.219,MAE:72.803
```

사이킷런 선형 회귀 모델 `LinearRegression`, `Ridge`, `Lasso` 모두 학습해 예측 성능 확인

- 모델과 학습/테스트 데이터셋 입력하면 성능 평가 수치 반환하는 `get_model_predict()` 함수 구현
- 원-핫 인코딩 적용 후 선형 회귀 예측 성능 많이 향상

```
coef=pd.Series(lr_reg.coef_,index=X_features_ohe.columns)
coef_sort=coef.sort_values(ascending=False)[:20]
sns.barplot(x=coef_sort.values,y=coef_sort.index)
```

```
<Axes: ylabel='None'>
```



원-핫 인코딩 데이터셋에서 회귀 계수 높은 피처 다시 시각화

- 회귀 계수 상위 20개 피처 추출
- 피처들 영향도 달라지고 모델 성능 향상된 것 확인

```
from sklearn.ensemble import RandomForestRegressor,GradientBoostingRegressor
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor

# 랜덤 포레스트, GBM, XGBoost, LightGBM model별로 평가 수행
rf_reg=RandomForestRegressor(n_estimators=500)
gbm_reg=GradientBoostingRegressor(n_estimators=500)
xgb_reg=XGBRegressor(n_estimators=500)
lgbm_reg=LGBMRegressor(n_estimators=500)

for model in [rf_reg,gbm_reg,xgb_reg,lgbm_reg]:
  # XGBoost의 경우 DataFrame이 입력될 경우 버전에 따라 오류 발생 가능. ndarray로 변환.
  get_model_predict(model,X_train.values,X_test.values,y_train.values,y_test.values,is_expm1=True)
```

```
### RandomForestRegressor ###
RMSLE:0.355,RMSE:50.355,MAE:31.203
### GradientBoostingRegressor ###
RMSLE:0.330,RMSE:53.347,MAE:32.746
### XGBRegressor ###
RMSLE:0.339,RMSE:51.475,MAE:31.357
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.002134 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 348
[LightGBM] [Info] Number of data points in the train set: 7620, number of used features: 72
[LightGBM] [Info] Start training from score 4.582043
### LGBMRegressor ###
RMSLE:0.319,RMSE:47.215,MAE:29.029
```

회귀 트리 이용해 회귀 예측 수행 => Target 값의 로그 변환된 값과 원-핫 인코딩된 피처 데이터셋 그대로 이용해 랜덤 포레스트, GBM, XGBoost, LightGBM 순차적으로 성능 평가

- 앞의 선형 회귀 모델보다 회귀 예측 성능 개선
- 회귀 트리가 선형 회귀보다 더 나은 성능 가진다는 의미는 X
- 데이터셋 유형에 따라 결과는 얼마든지 달라질 수 있음

(10) 회귀 실습 - 캐글 주택 가격: 고급 회귀 기법

> 캐글 house-prices-advanced-regression-techniques 데이터셋 활용
> => 미국 아이오와 주의 에임스 지방의 주택 가격 정보

데이터 사전 처리(Preprocessing)

```
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

house_df_org=pd.read_csv(r'house_price.csv')
house_df=house_df_org.copy()
house_df.head(3)
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	MoSold	YrSold	Sal
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	2	2008	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	5	2007	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	9	2008	

3 rows × 81 columns

house_price.csv 업로드 후 데이터 확인

- 원본 csv 파일 기반의 DataFrame은 보관, 복사해서 데이터 가공
- Target 값 => 맨 마지막 칼럼 SalePrice

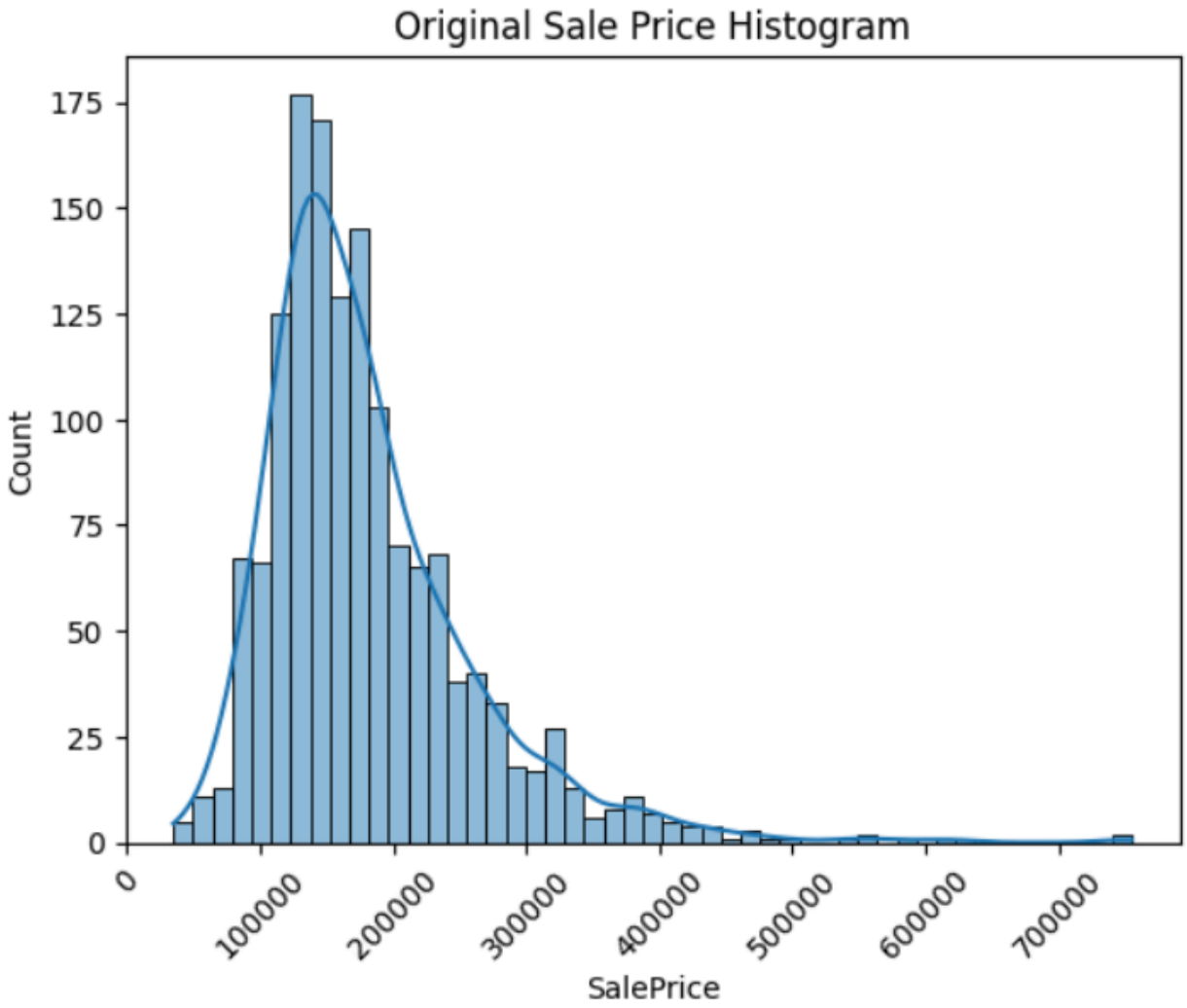
```
print('데이터 세트의 Shape:',house_df.shape)
print('₩n전체 피처의 type ₩n',house_df.dtypes.value_counts())
isnull_series=house_df.isnull().sum()
print('₩nNull 칼럼과 그 건수:₩n',isnull_series[isnull_series>0].sort_values(ascending=False))
```

```
데이터 세트의 Shape: (1460, 81)

전체 피처의 type
 object     43
int64      35
float64     3
Name: count, dtype: int64

Null 칼럼과 그 건수:
 PoolQC          1453
MiscFeature      1406
Alley            1369
Fence            1179
MasVnrType        872
FireplaceQu       690
LotFrontage       259
GarageType         81
GarageYrBlt        81
GarageFinish       81
GarageQual         81
GarageCond         81
BsmtExposure       38
BsmtFinType2       38
BsmtQual           37
```

데이터셋 전체 크기, 칼럼 타입, Null이 있는 칼럼과 그 건수 내림차순으로 출력

- 1460개 레코드와 81개 피처로 구성
- Null 값 너무 많은 피처는 드롭

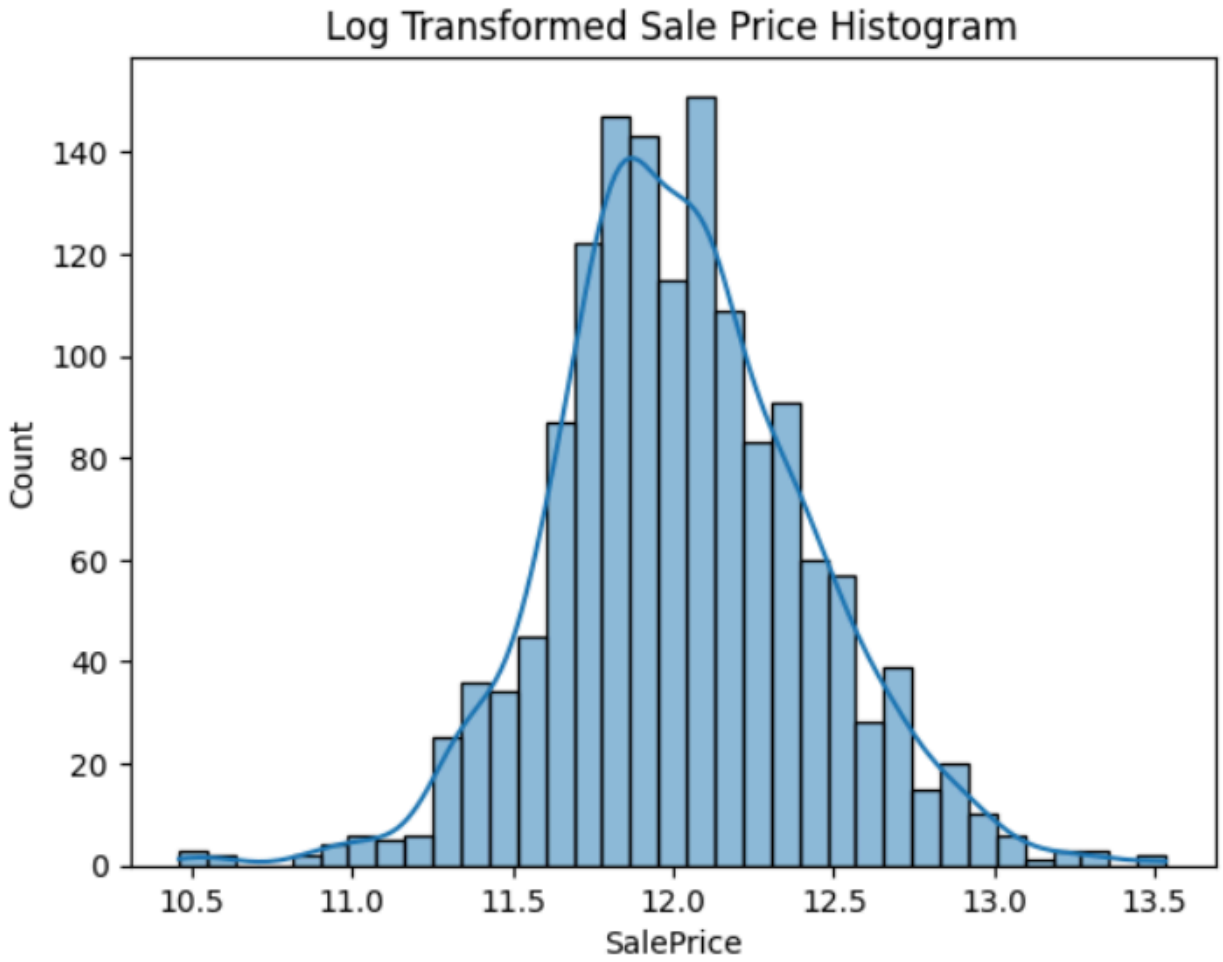
```
plt.title('Original Sale Price Histogram')
plt.xticks(rotation=45)
sns.histplot(house_df['SalePrice'],kde=True)
plt.show()
```



회귀 모델 적용 전, 타깃 값 분포도가 정규 분포인지 확인

- 데이터 값 분포가 정규 분포에서 벗어나 있음

```
plt.title('Log Transformed Sale Price Histogram')
log_SalePrice=np.log1p(house_df['SalePrice'])
sns.histplot(log_SalePrice,kde=True)
plt.show()
```



넘파이 `log1p()` 이용해 로그 변환하고 다시 분포도 확인

- 정규 분포 형태로 결괏값 분포

```
# SalePrice 로그 변환
original_SalePrice=house_df['SalePrice']
house_df['SalePrice']=np.log1p(house_df['SalePrice'])

# Null이 너무 많은 칼럼과 불필요한 칼럼 삭제
house_df.drop(['Id','PoolQC','MiscFeature','Alley','Fence','FireplaceQu'],axis=1,inplace=True,errors='ignore')
# 드롭하지 않는 숫자형 Null 칼럼은 평균값으로 대체
house_df.fillna(house_df.mean(numeric_only=True),inplace=True)

# Null 값이 있는 피처명과 타입을 추출
null_column_count=house_df.isnull().sum()[house_df.isnull().sum()>0]
print('## Null 피처의 Type :₩n',house_df.dtypes[null_column_count.index])
```

```
## Null 피처의 Type :
 MasVnrType      object
BsmtQual        object
BsmtCond        object
BsmtExposure    object
BsmtFinType1    object
BsmtFinType2    object
Electrical      object
GarageType      object
GarageFinish    object
GarageQual      object
GarageCond      object
dtype: object
```

로그 변환 및 Null 피처의 전처리 수행

- SalePrice 로그 변환한 뒤 DataFrame에 반영
- Null 값 많은 피처, 단순 식별자 삭제
- `DataFrame.fillna(DataFrame.mean())` => Null 값인 숫자형 피처만 평균값으로 대체

```
print('get_dummies() 수행 전 데이터 Shape:',house_df.shape)
house_df_ohe=pd.get_dummies(house_df)
print('get_dummies() 수행 후 데이터 Shape:',house_df_ohe.shape)

null_column_count=house_df_ohe.isnull().sum()[house_df_ohe.isnull().sum()>0]
print('## Null 피처의 Type :₩n',house_df_ohe.dtypes[null_column_count.index])
```

```
get_dummies() 수행 전 데이터 Shape: (1460, 75)
get_dummies() 수행 후 데이터 Shape: (1460, 270)
## Null 피처의 Type :
 Series([], dtype: object)
```

문자형 피처는 판다스 `get_dummies()` 이용해 원-핫 인코딩으로 변환

- 자동으로 문자열 피처를 원-핫 인코딩으로 변환하면서 Null 값은 모든 인코딩 값이 0으로 변환되는 방식으로 대체해줌 => 별도의 Null 값 대체하는 로직 필요 X

선형 회귀 모델 학습/예측/평가

```
def get_rmse(model):
  pred=model.predict(X_test)
  mse=mean_squared_error(y_test,pred)
  rmse=np.sqrt(mse)
  print(model.__class__.__name__,'로그 변환된 RMSE:',np.round(rmse,3))
  return rmse

def get_rmses(models):
  rmses=[]
  for model in models:
    rmse=get_rmse(model)
    rmses.append(rmse)
  return rmses
```

로그 변환된 RMSE 계산하는 함수 구현

- 실제 값도 로그 변환됐고 예측값도 이를 반영한 로그 변환 값 => 예측 결과 오류에 RMSE만 적용하면 RMSLE가 작동으로 측정
- `get_rmse(model)` : 단일 모델의 RMSE 값 반환
- `get_rmses(models)` : 여러 모델의 RMSE 값 반환

```
from sklearn.linear_model import LinearRegression,Ridge,Lasso
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

y_target=house_df_ohe['SalePrice']
X_features=house_df_ohe.drop('SalePrice',axis=1,inplace=False)
X_train,X_test,y_train,y_test=train_test_split(X_features,y_target,test_size=0.2,random_state=156)

# LinearRegression, Ridge, Lasso 학습, 예측, 평가
lr_reg=LinearRegression()
lr_reg.fit(X_train,y_train)
ridge_reg=Ridge()
ridge_reg.fit(X_train,y_train)
lasso_reg=Lasso()
lasso_reg.fit(X_train,y_train)

models=[lr_reg,ridge_reg,lasso_reg]
get_rmses(models)
```

```
LinearRegression 로그 변환된 RMSE: 0.132
Ridge 로그 변환된 RMSE: 0.127
Lasso 로그 변환된 RMSE: 0.176
[np.float64(0.13183184688250701),
 np.float64(0.1274058283626616),
 np.float64(0.17628250556471403)]
```

선형 회귀 모델 학습하고 예측, 평가

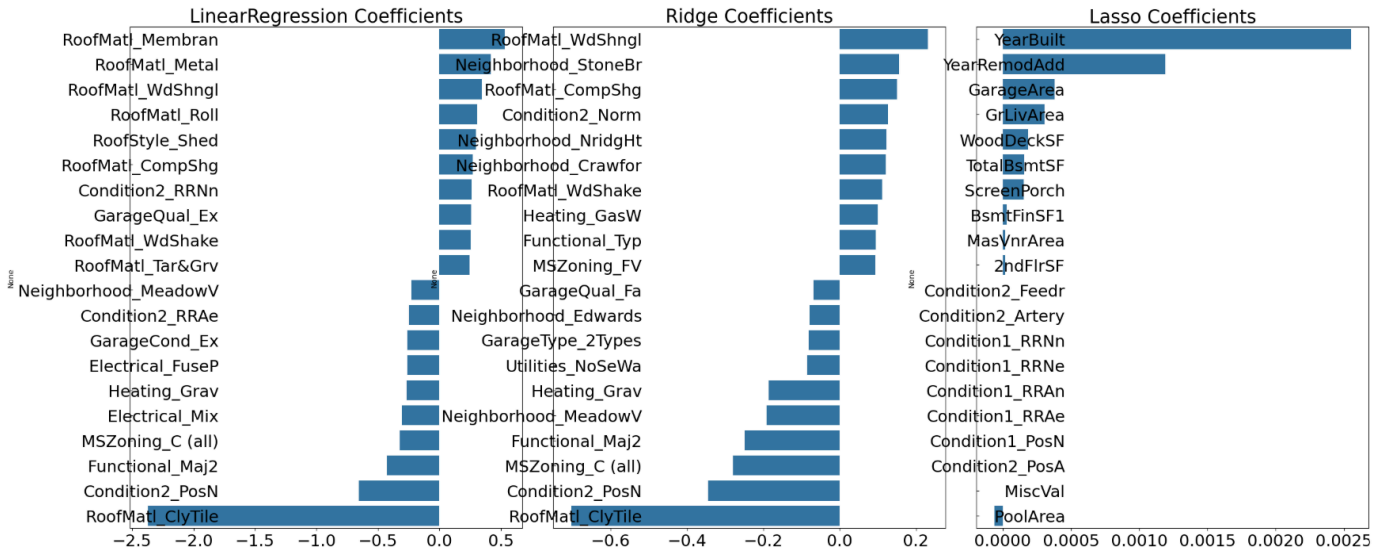
- 라쏘 회귀의 경우 회귀 성능이 타 회귀 방식보다 많이 떨어짐 => 최적 하이퍼 파라미터 튜닝 필요

```
def get_top_bottom_coef(model,n=10):
  # coef_ 속성을 기반으로 Series 객체를 생성. index는 칼럼명.
  coef=pd.Series(model.coef_,index=X_features.columns)

  # + 상위 10개, - 하위 10개의 회귀 계수를 추출해 반환.
  coef_high=coef.sort_values(ascending=False).head(n)
  coef_low=coef.sort_values(ascending=False).tail(n)
  return coef_high,coef_low
```

회귀 계수 값의 상위 10개, 하위 10개의 피처명과 그 회귀 계수 값 갖는 판다스 Series 객체 반환하는 함수 구현

```
def visualize_coefficient(models):
  # 3개 회귀 모델의 시각화를 위해 3개의 칼럼을 가지는 subplot 생성
  fig,axs=plt.subplots(figsize=(24,10),nrows=1,ncols=3)
  fig.tight_layout()
  # 입력 인자로 받은 list 객체인 models에서 차례로 model을 추출해 회귀 계수 시각화
  for i_num,model in enumerate(models):
    # 상위 10개, 하위 10개 회귀 계수를 구하고, 이를 판다스 concat으로 결합
    coef_high,coef_low=get_top_bottom_coef(model)
    coef_concat=pd.concat([coef_high,coef_low])
    # ax subplot에 barchar로 표현. 한 화면에 표현하기 위해 tick label 위치와 font 크기 조정
    axs[i_num].set_title(model.__class__.__name__+' Coefficients',size=25)
    axs[i_num].tick_params(axis='y',direction='in',pad=-120)
    for label in (axs[i_num].get_xticklabels()+axs[i_num].get_yticklabels()):
      label.set_fontsize(22)
    sns.barplot(x=coef_concat.values,y=coef_concat.index,ax=axs[i_num])

# 앞 예제에서 학습한 lr_reg, ridge_reg,lasso_reg 모델의 회귀 계수 시각화
```

```
models=[lr_reg,ridge_reg,lasso_reg]
visualize_coefficient(models)
```



생성한 `get_top_bottom_coef(model,n=10)` 함수 이용해 모델별 회귀 계수 시각화

- 시각한 위한 함수로 `visualize_coefficient(models)` 생성
- list 객체로 모델 입력받아 모델별로 회귀 계수 상위 10개, 하위 10개 추출해 가로 막대 그래프 형태로 출력
- OLS 기반의 LinearRegression과 Ridge의 경우는 회귀 계수가 유사한 형태로 분포
- BUT 라쏘(Lasso)는 전체적으로 회귀 계수 값 매우 작고 그중 YearBuilt가 가장 크고 다른 피처의 회귀 계수는 너무 작음

```
from sklearn.model_selection import cross_val_score

def get_avg_rmse_cv(models):
  for model in models:
    # 분할하지 않고 전체 데이터로 cross_val_score() 수행, 모델별 CV RMSE 값과 평균 RMSE 출력
    rmse_list=np.sqrt(-cross_val_score(model,X_features,y_target,scoring='neg_mean_squared_error',cv=5))
    rmse_avg=np.mean(rmse_list)
    print('₩n{0} CV RMSE 값 리스트: {1}'.format(model.__class__.__name__,np.round(rmse_list,3)))
    print('{0} CV 평균 RMSE 값: {1}'.format(model.__class__.__name__,np.round(rmse_avg,3)))

# 앞 예제에서 학습한 ridge_reg, lasso_reg 모델의 CV RMSE 값 출력
models=[ridge_reg,lasso_reg]
get_avg_rmse_cv(models)
```

```
Ridge CV RMSE 값 리스트: [0.117 0.154 0.142 0.117 0.189]
Ridge CV 평균 RMSE 값: 0.144

Lasso CV RMSE 값 리스트: [0.161 0.204 0.177 0.181 0.265]
Lasso CV 평균 RMSE 값: 0.198
```

학습&테스트 데이터셋을 `train_test_split()` 으로 분할하지 않고 전체 데이터 세트인 `X_features` 와 `y_target` 을 5개의 교차 검증 폴드 세트로 분할해 평균 RMSE 측정

- cross_val_score() 이용
- 여전히 라쏘가 릿지 모델보다 성능 떨어짐

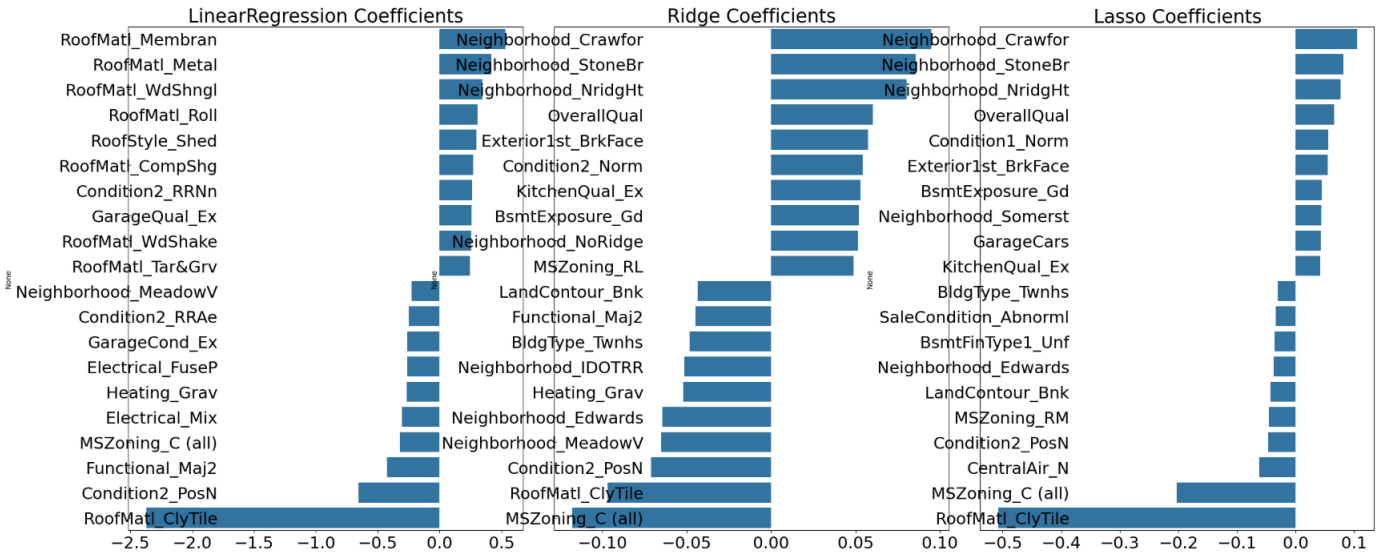
```
from sklearn.model_selection import GridSearchCV

def print_best_params(model,params):
  grid_model=GridSearchCV(model,param_grid=params,scoring='neg_mean_squared_error',cv=5)
  grid_model.fit(X_features,y_target)
  rmse=np.sqrt(-1*grid_model.best_score_)
  print('{0} 5 CV 시 최적 평균 RMSE 값:{1}, 최적 alpha:{2}'.format(model.__class__.__name__,np.round(rmse,4),grid_model.best_params_))

ridge_params={'alpha':[0.05,0.1,1,5,8,10,12,15,20]}
lasso_params={'alpha':[0.001,0.005,0.008,0.05,0.03,0.1,0.5,1,5,10]}
print_best_params(ridge_reg,ridge_params)
print_best_params(lasso_reg,lasso_params)
```

```
Ridge 5 CV 시 최적 평균 RMSE 값:0.1418, 최적 alpha:{'alpha': 12}
Lasso 5 CV 시 최적 평균 RMSE 값:0.142, 최적 alpha:{'alpha': 0.001}
```

릿지와 라쏘 모델에 대해 alpha 하이퍼 파라미터 변화시키면서 최적 값 도출

- 모델별로 최적화 하이퍼 파라미터 작업을 반복적으로 진행하므로 이를 위한 별도의 함수 생성
- print_best_params(model,params) : 모델과 하이퍼 파라미터 딕셔너리 객체 받아 최적화 작업의 결과 표시
- alpha 값 최적화 이후 라쏘 모델의 예측 성능 많이 좋아진 것 확인

```
# 앞의 최적화 alpha 값으로 학습 데이터로 학습, 테스트 데이터로 예측 및 평가 수행
lr_reg=LinearRegression()
lr_reg.fit(X_train,y_train)
ridge_reg=Ridge(alpha=12)
ridge_reg.fit(X_train,y_train)
lasso_reg=Lasso(alpha=0.001)
lasso_reg.fit(X_train,y_train)

# 모든 모델의 RMSE 출력
models=[lr_reg,ridge_reg,lasso_reg]
get_rmses(models)

# 모든 모델의 회귀 계수 시각화
models=[lr_reg,ridge_reg,lasso_reg]
visualize_coefficient(models)
```



선형 모델에 최적 alpha 값 설정한 뒤, `train_test_split()` 으로 분할된 학습 데이터와 테스트 데이터 이용해 모델의 학습/예측/평가 수행하고 모델별 회귀 계수 시각화

- alpha 값 최적화 후 테스트 데이터셋의 예측 성능 더 좋아짐
- 기존에는 라쏘 모델의 회귀 계수가 나머지 두 모델과 많은 차이 있었지만, 이번에는 릿지와 라쏘 모델에서 비슷한 회귀 계수가 높음
- 다만 라쏘 모델의 경우 릿지에 비해 동일한 피처라도 회귀 계수 값 상당히 작음

```
from scipy.stats import skew

# object가 아닌 숫자형 피처의 칼럼 index 객체 추출
features_index=house_df.dtypes[house_df.dtypes!='object'].index
# house_df에 칼럼 index를 []로 입력하면 해당하는 칼럼 데이터 세트 반환. apply lambda로 skew() 호출
skew_features=house_df[features_index].apply(lambda x:skew(x))
# skew(왜곡) 정도가 1 이상인 칼럼만 추출
skew_features_top=skew_features[skew_features>1]
print(skew_features_top.sort_values(ascending=False))
```

```
MiscVal         24.451640
PoolArea        14.813135
LotArea         12.195142
3SsnPorch       10.293752
LowQualFinSF     9.002080
KitchenAbvGr     4.483784
BsmtFinSF2       4.250888
ScreenPorch      4.117977
BsmtHalfBath     4.099186
EnclosedPorch    3.086696
MasVnrArea       2.673661
LotFrontage      2.382499
OpenPorchSF      2.361912
BsmtFinSF1       1.683771
WoodDeckSF       1.539792
TotalBsmtSF      1.522688
```

피처 데이터셋 분포도로 왜곡된 정도 확인

- 사이파이 stats 모듈의 `skew()` 함수 이용 => 숫자형 피처의 칼럼 index 객체 추출해 구한 숫자형 칼럼 데이터셋의 apply lambda 식 `skew()` 호출해 숫자형 피처의 왜곡 정도 구함
- `skew()` 적용하는 숫자형 피처에서 원-핫 인코딩된 카테고리 숫자형 피처는 제외 (카테고리 피처는 코드성 피처이므로 인코딩 시 왜곡될 가능성 높음) => 원-핫 인코딩

적용된 house_df_ohe 가 아니라 원-핫 인코딩 적용되지 않은 house_df 에 skew() 함수 적용

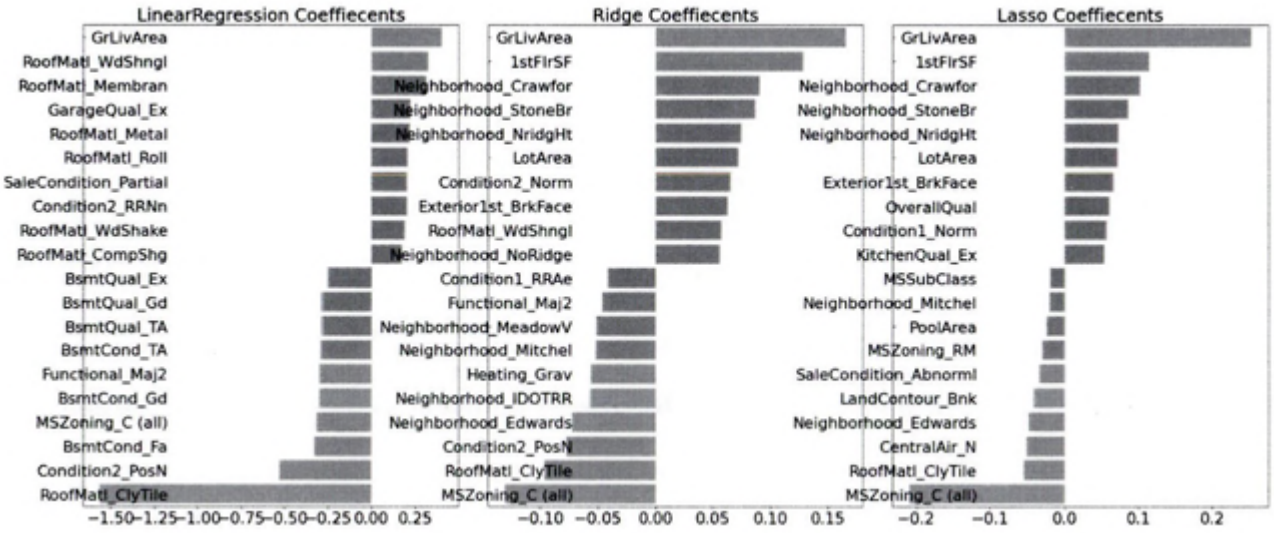
```
house_df[skew_features_top.index]=np.log1p(house_df[skew_features_top.index])
```

추출된 왜곡 정도가 높은 피처를 로그 변환

```
# 왜곡 정도가 높은 피처를 로그 변환했으므로 다시 원-핫 인코딩을 적용하고 피처/타깃 데이터 세트 생성
house_df_ohe=pd.get_dummies(house_df)
y_target=house_df_ohe['SalePrice']
X_features=house_df_ohe.drop('SalePrice',axis=1,inplace=False)
X_train,X_test,y_train,y_test=train_test_split(X_features,y_target,test_size=0.2,random_state=156)

# 피처를 로그 변환한 후 다시 최적 하이퍼 파라미터와 RMSE 출력
ridge_params={'alpha':[0.05,0.1,1,5,8,10,12,15,20]}
lasso_params={'alpha':[0.001,0.005,0.008,0.05,0.03,0.1,0.5,1,5,10]}
print_best_params(ridge_reg,ridge_params)
print_best_params(lasso_reg,lasso_params)
```

```
Ridge 5 CV 시 최적 평균 RMSE 값:0.1275, 최적 alpha:{'alpha': 10}
Lasso 5 CV 시 최적 평균 RMSE 값:0.1252, 최적 alpha:{'alpha': 0.001}
```

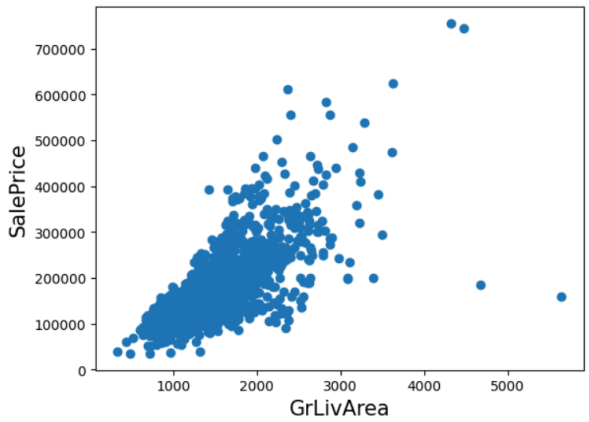
다시 원-핫 인코딩 적용한 house_df_ohe 와 이에 기반한 피처 데이터셋, 타깃 데이터셋, 학습/테스트 데이터셋 다시 만들고 print_best_params() 함수 이용해 최적 alpha 값과 RMSE 출력



위의 train_test_split() 으로 분할된 학습&테스트 데이터 이용해 모델의 학습/예측/평가 및 모델별 회귀 계수 시각화한 결과

- 세 모델 모두 GrLivArea(주거 공간 크기)가 회귀 계수 가장 높은 피처

```
plt.scatter(x=house_df_org['GrLivArea'],y=house_df_org['SalePrice'])
plt.ylabel('SalePrice',fontsize=15)
plt.xlabel('GrLivArea',fontsize=15)
plt.show()
```



주택 가격 데이터 변환 이전 원본 데이터셋 house_df_org에서 GrLivArea와 타깃 값 SalePrice의 관계 시각화

- 일반적으로 주거 공간 큰 집일수록 가격 비쌈 => GrLivArea 피처는 SalePrice와 양의 상관도 매우 높음
- BUT 일반적 관계에서 너무 어긋난 outlier 존재 => GrLivArea가 4000 평방피트 이상임에도 가격이 500000 달러 이하인 데이터는 모두 이상치로 간주 후 삭제 필요

```
# GrLivArea와 SalePrice 모두 로그 변환됐으므로 이를 반영한 조건 생성
cond1=house_df_ohe['GrLivArea']>np.log1p(4000)
cond2=house_df_ohe['SalePrice']<np.log1p(500000)
outlier_index=house_df_ohe[cond1&cond2].index

print('이상치 레코드 idnex :',outlier_index.values)
print('이상치 삭제 전 house_df_ohe shape :',house_df_ohe.shape)

# DataFrame의 인덱스를 이용해 이상치 레코드 삭제
house_df_ohe.drop(outlier_index,axis=0,inplace=True)
print('이상치 삭제 후 house_df_ohe shape:',house_df_ohe.shape)
```

```
이상치 레코드 idnex : [ 523 1298]
이상치 삭제 전 house_df_ohe shape : (1460, 270)
이상치 삭제 후 house_df_ohe shape: (1458, 270)
```

데이터 변환 모두 완료된 house_df_ohe에서 대상 데이터 필터링

- GrLivArea와 SalePrice의 로그 변환 반영한 조건 생성 후 불린 인덱싱으로 대상 찾아 DataFrame 인덱스와 `drop()` 이용해 해당 데이터 삭제

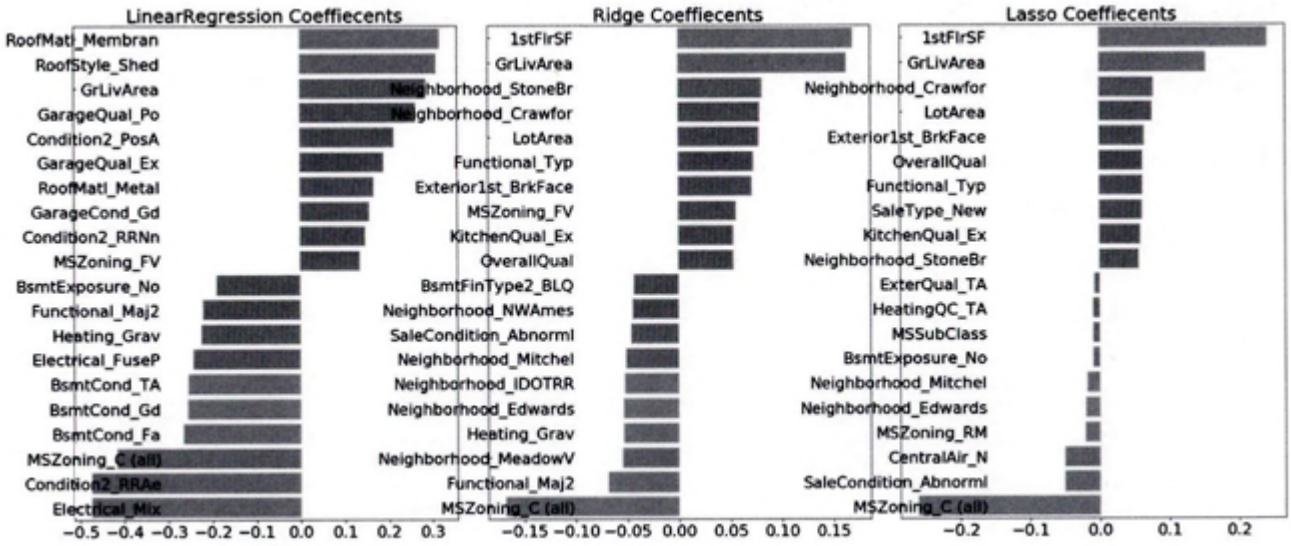
```
y_target=house_df_ohe['SalePrice']
X_features=house_df_ohe.drop('SalePrice',axis=1,inplace=False)
X_train,X_test,y_train,y_test=train_test_split(X_features,y_target,test_size=0.2,random_state=156)

ridge_params={'alpha':[0.05,0.1,1,5,8,10,12,15,20]}
lasso_params={'alpha':[0.001,0.005,0.008,0.05,0.03,0.1,0.5,1,5,10]}
print_best_params(ridge_reg,ridge_params)
print_best_params(lasso_reg,lasso_params)
```

```
Ridge 5 CV 시 최적 평균 RMSE 값:0.1125, 최적 alpha:{'alpha': 8}
Lasso 5 CV 시 최적 평균 RMSE 값:0.1122, 최적 alpha:{'alpha': 0.001}
```

업데이트된 house_df_ohe 기반으로 피처&타깃 데이터셋 다시 생성, `print_best_params()` 함수 이용해 릿지&라쏘 모델의 최적화 수행하고 결과 출력

- 단 두 개의 이상치 데이터만 제거했는데 예측 수치 크게 향상된 것 확인
- GrLivArea 속성이 회귀 모델에서 차지하는 영향도 크기에 이상치 개선이 성능 개선에 큰 의미 가짐

> 보통 ML 프로세스 중 데이터 가공은 알고리즘 적용 이전에 수행
> BUT ML 알고리즘 적용 이전에 데이터 선처리 작업 완벽 수행하란 의미는 X
> => 대략의 데이터 가공과 모델 최적화 수행 후 다시 이에 기반한 여러 기법의 데이터 가공과 하이퍼 파라미터 기반 모델 최적화 반복 수행하는 것이 바람직한 ML 모델 생성 과정!



이상치 제거된 데이터셋 기반으로 다시 `train_test_split()` 로 분할된 데이터셋의 RMSE 수치 및 회귀 계수 시각화한 결과

회귀 트리 모델 학습/예측/평가

```
from xgboost import XGBRegressor

xgb_params={'n_estimators':[1000]}
xgb_reg=XGBRegressor(n_estimators=1000,learning_rate=0.05,colsample_bytree=0.5,subsample=0.8)
print_best_params(xgb_reg,xgb_params)
```

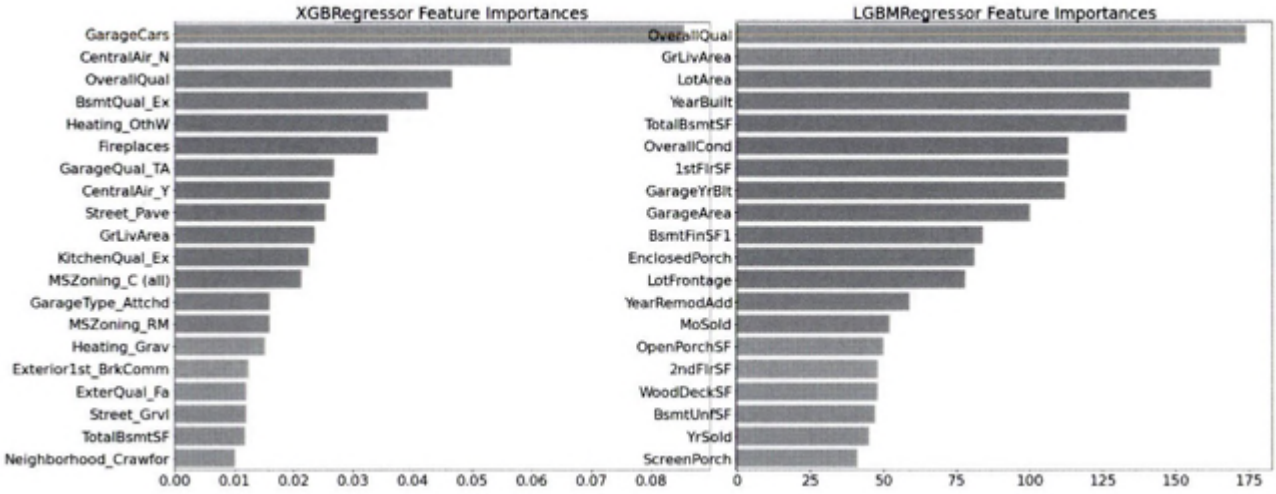
```
XGBRegressor 5 CV 시 최적 평균 RMSE 값:0.1206, 최적 alpha:{'n_estimators': 1000}
```

```
from lightgbm import LGBMRegressor

lgbm_params={'n_estimators':[1000]}
lgbm_reg=LGBMRegressor(n_estimators=1000,learning_rate=0.05,num_leaves=5,subsample=0.6,colsample_bytree=0.4,reg_lambda=10,n_jobs=1)
print_best_params(lgbm_reg,lgbm_params)
```

```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.001340 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 3155
[LightGBM] [Info] Number of data points in the train set: 1166, number of used features: 171
[LightGBM] [Info] Start training from score 12.021352
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.001049 seconds.
```

회귀 트리 이용해 회귀 모델 생성

- XGBoost는 `XGBRegressor` 클래스, LightGBM은 `LGBMRegressor` 클래스 이용
- 하이퍼 파라미터 설정 미리 적용한 상태로 5 폴드 세트에 대한 평균 RMSE 값 구하기



모델의 피처 중요도 시각화 결과

회귀 모델의 예측 결과 혼합을 통한 최종 예측

```
def get_rmse_pred(preds):
  for key in preds.keys():
    pred_value=preds[key]
    mse=mean_squared_error(y_test,pred_value)
    rmse=np.sqrt(mse)
    print('{0} 모델의 RMSE: {1}'.format(key,rmse))

# 개별 모델의 학습
ridge_reg=Ridge(alpha=8)
ridge_reg.fit(X_train,y_train)
lasso_reg=Lasso(alpha=0.001)
lasso_reg.fit(X_train,y_train)
# 개별 모델 예측
ridge_pred=ridge_reg.predict(X_test)
lasso_pred=lasso_reg.predict(X_test)

# 개별 모델 예측값 혼합으로 최종 예측값 도출
pred=0.4*ridge_pred+0.6*lasso_pred
preds={'최종 혼합':pred,'Ridge':ridge_pred,'Lasso':lasso_pred}

# 최종 혼합 모델, 개별 모델의 RMSE 값 출력
get_rmse_pred(preds)
```

```
최종 혼합 모델의 RMSE: 0.10006075517615193
Ridge 모델의 RMSE: 0.10340697165289348
Lasso 모델의 RMSE: 0.10024171179335342
```

개별 회귀 모델의 예측 결괏값 혼합해 이를 기반으로 최종 회귀 값 예측

- 최종 혼합 모델, 개별 모델의 RMSE 값 출력하는 `get_rmse_pred()` 함수 생성하고 각 모델 예측값 계산한 뒤 개별 모델과 최종 혼합 모델의 RMSE 구하기
- 최종 혼합 모델의 RMSE가 개별 모델보다 성능 면에서 약간 개선된 것 확인

```
xgb_reg=XGBRegressor(n_estimators=1000,learning_rate=0.05,colsample_bytree=0.5,subsample=0.8)
lgbm_reg=LGBMRegressor(n_estimators=1000,learning_rate=0.05,num_leaves=4,subsample=0.6,colsample_bytree=0.4,reg_lambda=10,n_jobs=-1)
xgb_reg.fit(X_train,y_train)
lgbm_reg.fit(X_train,y_train)
xgb_pred=xgb_reg.predict(X_test)
lgbm_pred=lgbm_reg.predict(X_test)

pred=0.5*xgb_pred+0.5*lgbm_pred
preds={'최종 혼합':pred,'XGBM':xgb_pred,'LGBM':lgbm_pred}

get_rmse_pred(preds)
```

```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.000806 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 3174
[LightGBM] [Info] Number of data points in the train set: 1166, number of used features: 172
[LightGBM] [Info] Start training from score 12.025343
최종 혼합 모델의 RMSE: 0.10215383402419025
XGBM 모델의 RMSE: 0.10761344291735733
LGBM 모델의 RMSE: 0.10363891833477148
```

XGBoost와 LightGBM 혼합해 결과 확인

- 혼합 모델의 RMSE가 개별 모델의 RMSE보다 조금 향상된 것 확인

스태킹 앙상블 모델을 통한 회귀 예측

> <스태킹 모델의 핵심>
> 여러 개별 모델의 예측 데이터 각각 스태킹 형태로 결합해 최종 메타 모델의 학습용/

> 테스트용 피처 데이터셋 만드는 것

```
from sklearn.model_selection import KFold
from sklearn.metrics import mean_absolute_error

# 개별 기반 모델에서 최종 메타 모델이 사용할 학습 및 테스트용 데이터를 생성하기 위한 함수
def get_stacking_base_datasets(model,X_train_n,y_train_n,X_test,n,n_folds):
  # 지정된 n_folds 값으로 KFold 생성
  kf=KFold(n_splits=n_folds,shuffle=False)
  # 추후에 메타 모델이 사용할 학습 데이터 반환을 위한 넘파이 배열 초기화
  train_fold_pred=np.zeros((X_train_n.shape[0],1))
  test_pred=np.zeros((X_test_n.shape[0],n_folds))
  print(model.__class__.__name__,'model 시작')

  for folder_counter,(train_index,valid_index) in enumerate(kf.split(X_train_n)):
    # 입력된 학습 데이터에서 기반 모델이 학습/예측할 폴드 데이터 세트 추출
    print('\t 폴드 세트:',folder_counter,'시작')
    X_tr=X_train_n[train_index]
    y_tr=y_train_n[train_index]
    X_te=X_train_n[valid_index]

    # 폴드 세트 내부에서 다시 만들어진 학습 데이터로 기반 모델의 학습 수행
    model.fit(X_tr,y_tr)
    # 폴드 세트 내부에서 다시 만들어진 검증 데이터로 기반 모델 예측 후 데이터 저장
    train_fold_pred[valid_index,:]=model.predict(X_te).reshape(-1,1)
    # 입력된 원본 테스트 데이터를 폴드 세트 내 학습된 기반 모델에서 예측 후 데이터 저장
    test_pred[:,folder_counter]=model.predict(X_test_n)

  # 폴드 세트 내에서 원본 테스트 데이터를 예측한 데이터를 평균하여 테스트 데이터로 생성
  test_pred_mean=np.mean(test_pred,axis=1).reshape(-1,1)

  # train_fold_pred는 최종 메타 모델이 사용하는 학습 데이터, test_pred_mean은 테스트 데이터
  return train_fold_pred,test_pred_mean
```

개별 모델을 스태킹 모델로 제공하기 위해 데이트셋 생성 (4장 참고)

- `get_stacking_base_datasetss()` : 인자로 개별 기반 모델, 원래 사용되는 학습&테스트용 피처 데이터 입력받음
- 함수 내에서 개별 모델이 K-폴드 세트로 설정된 폴드 세트 내부에서 원본의 학습 데이터 다시 추출해 학습&예측 수행한 뒤 결과 저장
- 저장된 예측 데이터는 추후 메타 모델의 학습 피처 데이터셋으로 이용
- 함수 내에서 폴드 세트 내부 학습 데이터로 학습된 개별 모델이 인자로 입력된 원본 테스트 데이터셋 예측한 뒤, 예측 결과 평균해 테스트 데이터로 생성

```
# get_stacking_base_datasets()는 넘파이 ndarray를 인자로 사용하므로 DataFrame을 넘파이로 변환
X_train_n=X_train.values
X_test_n=X_test.values
y_train_n=y_train.values

# 각 개별 기반(Base) 모델이 생성한 학습용/테스트용 데이터 반환
ridge_train,ridge_test=get_stacking_base_datasets(ridge_reg,X_train_n,y_train_n,X_test_n,5)
lasso_train,lasso_test=get_stacking_base_datasets(lasso_reg,X_train_n,y_train_n,X_test_n,5)
xgb_train,xgb_test=get_stacking_base_datasets(xgb_reg,X_train_n,y_train_n,X_test_n,5)
lgbm_train,lgbm_test=get_stacking_base_datasets(lgbm_reg,X_train_n,y_train_n,X_test_n,5)
```

```
Ridge  model 시작
Lasso  model 시작
XGBRegressor  model 시작
LGBMRegressor  model 시작
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.000889 seconds
```

`get_stacking_base_datasets()` 모델별로 적용해 메타 모델이 사용할 학습&테스트 피처 데이터셋 추출

- 적용할 개별 모델: 릿지, 라쏘, XGBoost, LightGBM

```
# 개별 모델이 반환한 학습 및 테스트용 데이터 세트를 스태킹 형태로 결합
Stack_final_X_train=np.concatenate((ridge_train,lasso_train,xgb_train,lgbm_train),axis=1)
Stack_final_X_test=np.concatenate((ridge_test,lasso_test,xgb_test,lgbm_test),axis=1)

# 최종 메타 모델은 라쏘 모델을 적용
meta_model_lasso=Lasso(alpha=0.0005)

# 개별 모델 예측값을 기반으로 새롭게 만들어진 학습/테스트 데이터로 메타 모델 예측 및 RMSE 측정
meta_model_lasso.fit(Stack_final_X_train,y_train)
final=meta_model_lasso.predict(Stack_final_X_test)
mse=mean_squared_error(y_test,final)
rmse=np.sqrt(mse)
print('스태킹 회귀 모델의 최종 RMSE 값은:',rmse)
```

```
스태킹 회귀 모델의 최종 RMSE 값은: 0.09685270532069315
```

각 개별 모델이 반환하는 학습용&테스트용 피처 데이터셋 결합해 최종 메타 모델에 적용

- 메타 모델은 별도의 라쏘 모델 이용, 최종적으로 예측 및 RMSE 측정
- 테스트 데이터셋에서 RMSE가 현재까지 가장 좋은 성능 평가 보임
- 스태킹 모델은 분류뿐만 아니라 회귀에서 특히 효과적으로 사용될 수 있는 모델

진규빈

이전 포스트
[파머완] 05 회귀