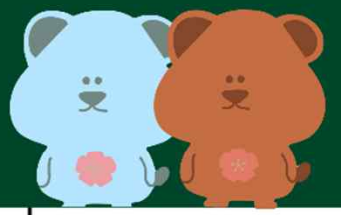
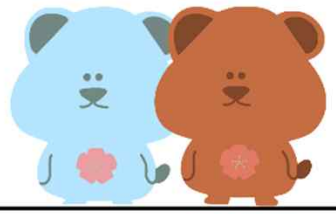


6주차 발표

김도하 남유림 여유진

EWHA EURON

목차



EWHA EURON

01 자전거 대여 예측

EWHA EURON

1.1 대회 소개

※ 자전거 대여 시스템이란?

- 도시 전역의 키오스크 위치 네트워크를 통해 회원 가입, 대여 및 자전거 반환 프로세스가 자동화 되는 자전거를 대여하는 수단.

- 전 세계 500개 이상의 자전거 대여 프로그램이 있음.

⇒ 자전거 대여 시스템에서 생성된 데이터는 여행 기간, 출발 위치, 도착 위치 및 경과 시간이
명시적으로 기록되기 때문에 센서 네트워크로서 기능하며, 이는 도시의 이동성을 연구하는 데 사용 가능.

※ 분석 방법

- 학습 데이터 세트를 이용해 선형 회귀와 트리 기반 회귀를 비교

1.2 Data Description

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

1.2 Data Description

(1) 데이터 로드 및 확인

```
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("bike_train.csv")
print(bike_df.shape)
bike_df.head(3)
```

```
(10886, 12)
```

	datetime	season	holiday	workingday	weather	temp	atemp	humidity
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80

```
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
```

(2) datetime 칼럼 가공(년, 월, 일, 시간)

```
# 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	w
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	

(3) 불필요한 칼럼 삭제

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

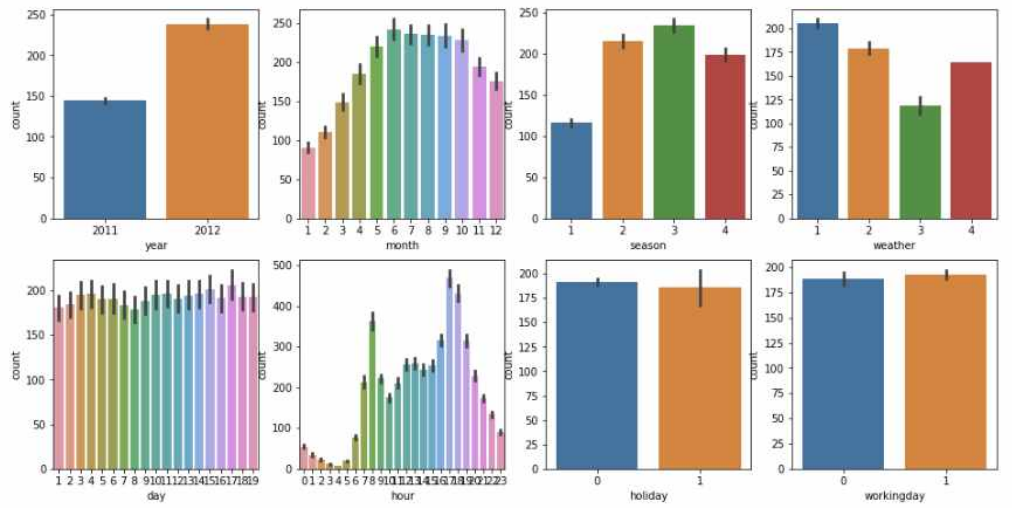
- year, month, day, hour 칼럼이 추가됐으므로, datetime 칼럼 필요 없음.
- count = casual + registered 이므로 casual과 registered가 따로 필요하지 않음. (오히려 상관도가 높아 예측을 저해할 우려가 있음)

EWHA EURON

1.2 Data Description

(4) 주요 칼럼별로 count가 어떻게 분포되는지 시각화

```
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])
```



- 년도 별 count를 보면 2012sus이 2011년보다 높음.
- 월별의 경우 1,2,3월이 낮고 6,7,8,9월이 높음.
- 계절을 보면, 봄, 겨울이 낮고 여름, 가을이 높음.
- 날씨의 경우 눈 또는 비가 있는 경우가 낮고 맑거나 약간 안개가 있는 경우가 높음.
- 시간의 경우 오전 출근 시간과 오후 퇴근 시간이 상대적으로 높음.
- 일자 간은 차이가 크지 않음.
- 휴일 여부 또는 주중 여부는 주중일 경우가 상대적으로 약간 높음.

1.3 RMSLE

MSE	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	- 회귀 모델의 주요 손실 함수. - 예측 값과 실제 값의 차이인 오차들의 제곱 평균으로 정의함. - 제곱을 하기 때문에 outlier에 민감함.
MAE	$\frac{1}{n}\sum_{i=1}^{n}(\lvert y_i - \hat{y}_i \rvert)$	- 실제 값과 예측 값의 차이인 오차들의 절대값 평균 - MSE보다는 outlier에 덜 민감함.
RMSE	$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$	- MSE에 root를 씌운 값. - 오류 지표를 실제 값과 유사한 단위로 다시 변환하기에 해석이 다소 용이함
RMSLE	$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(\hat{y}_i + 1) - log(y_i + 1))^2}$	- 오차를 구할 때, RMSE와는 log를 추가하는 점이 다름.

※ MSE, MAE, RMSE, RMSLE는 모두 값이 0에 가까울수록 좋은 성능

EWHA EURON

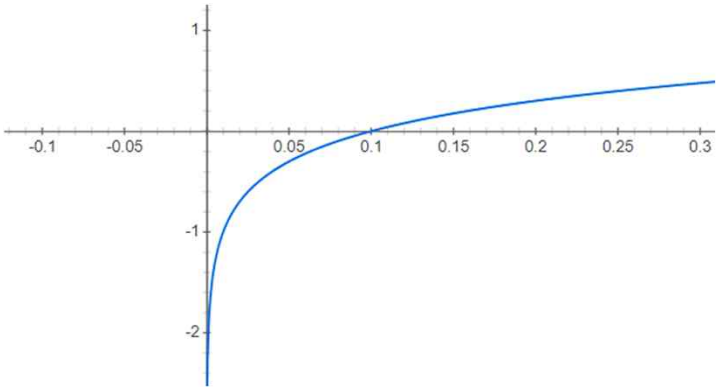
1.3 RMSLE

※ RMLSE의 특징

- outlier에 덜 민감함.
  (outlier가 있더라도 값의 변동폭이 크지 않음)

- 상대적 Error를 측정함.
  (값의 절대적 크기가 커지면 RSME의 값도 커지지만, RMSLE는 상대적 크기가 동일하다면 RMSLE의 값도 동일함)

- Under Estimation에 큰 패널티를 부여함.
  (그래프를 보면, 예측값이 실제값보다 작은 경우
  더 큰 패털티를 부여하고 있음)



1.4 성능 평가 함수

RMSLE, MSE, RMSE 평가 함수

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

# log 값 변환 시 NaN등의 이슈로 log() 가 아닌 log1p() 를 이용하여 RMSLE 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 이용하여 RMSE 계산
def rmse(y,pred):
    return np.sqrt(mean_squared_error(y,pred))

# MSE, RMSE, RMSLE 를 모두 계산
def evaluate_regr(y,pred):
    rmsle_val = rmsle(y,pred)
    rmse_val = rmse(y,pred)
    # MAE 는 scikit learn의 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))
```

※ RMSLE 함수를 만들 때, 주의점

Rmlse를 구할 때, 넘파이의 log( )함수를 이용하거나 사이킷런의 mean_squared_log_error( )를 이용할 수도 있지만, 데이터 값의 크기에 따라 오버플로/ 언더플로 오류가 발생하기 쉬움.

⇒ Log( ) 보다는 log1p()를 이용
log1p( )의 경우 1_log( ) 값으로 log 변환값에 1을 더하므로 오버플로/ 언더플로 오류와 같은 문제를 해결해줌.

⇒ Log1p( )로 변환된 값은 다시 넘파이의 expm1( )함수로 쉽게 원래 스케일로 복원 가능.

1.5 로그 변환, 피처 인코딩과 모델 학습/예측/평가

(1) LinearRegression 객체를 이용해 회귀 예측

```
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
```

(2) 오류 값 확인

```
def get_top_error_data(y_test, pred, n_tops = 5):
    # DataFrame에 컬럼들로 실제 대여횟수(count)와 예측 값을 서로 비교 할 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 실제값이 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)
```

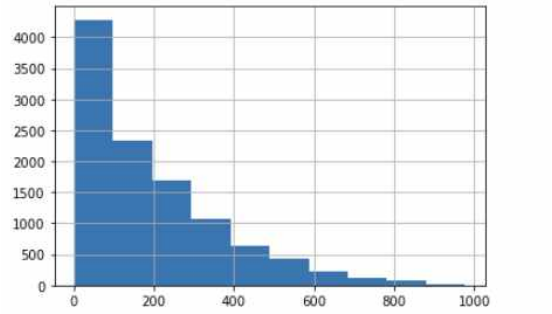
```
      real_count  predicted_count   diff
1618         890            322.0  568.0
3151         798            241.0  557.0
966          884            327.0  557.0
412          745            194.0  551.0
2817         856            310.0  546.0
```

※ 회귀에서 큰 예측 오류가 발생할 경우 가장 먼저 Target 값의 분포가 왜곡된 형태를 이루고 있는지 확인! (정규 분포 형태가 가장 좋음)

(3) count 칼럼 분포 형태 확인

```
y_target.hist()
```

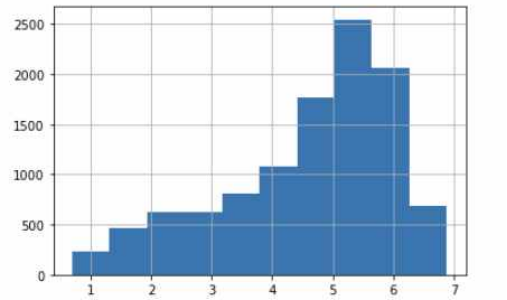
```
<AxesSubplot:>
```



(4) 왜곡된 값을 정규 분포 형태로 변환

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

```
<AxesSubplot:>
```



- 왜곡된 값을 정규 분포 형태로 바꾸는 가장 일반적인 방법은 로그를 적용해 변환
- 변경된 Target 값을 기반으로 학습하고 예측한 값은 다시 expm1( ) 함수를 적용해 원래 scale 값으로 원상 복구하면 됨.

EWHA EURON

1.5 로그 변환, 피처 인코딩과 모델 학습/예측/평가

(5) 변환 값을 이용해 학습 및 평가

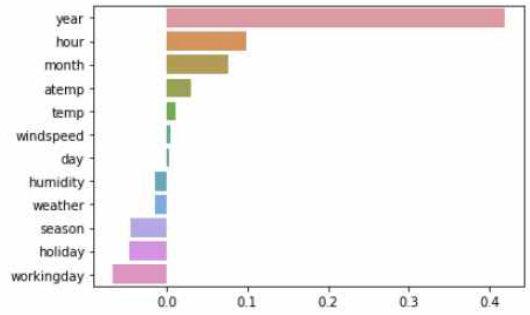
```
# 타깃 칼럼인 count 값을 log1p로 로그 변환
y_target_log = np.log1p(y_target)

# 로그 변환된 y_target_log를 반영하여 학습/테스트 데이터 셋 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 셋의 Target 값은 Log 변환되었으므로 다시 expm1를 이용하여 원래 scale로 변환
y_test_exp = np.expm1(y_test)

# 예측 값 역시 Log 변환된 타깃 기반으로 학습되어 예측되었으므로 다시 exmp1으로 scale변환
pred_exp = np.expm1(pred)

evaluate_regr(y_test_exp ,pred_exp)
```

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

(6) 각 피처의 회귀 계수 값 시각화

```
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)
```

```
<AxesSubplot:>
```



- year, hour, month, season, holiday, workingday 피처들의 회귀 계수가 상대적으로 높음.
- 이들 피처들의 경우 숫자 값 형태로 의미를 담고 있지만, 개별 숫자 값의 크기가 의미가 있는 것은 아님. 숫자 값으로 표현되었지만, 모두 카테고리형 피처임.

⇒ 이처럼 숫자형 카테고리 값을 선형 회귀에 사용할 경우 회귀 계수를 연산할 때 크게 영향을 받는 경우가 발생함.

⇒ 원-핫 인코딩을 적용해 변환해야 함.

1.5 로그 변환, 피처 인코딩과 모델 학습/예측/평가

(7) One Hot Encoding

```
# 'year', month', 'day', hour'등의 피처들을 One Hot Encoding
X_features_ohe = pd.get_dummies(X_features, columns=['year', 'month','day',
                                                     'hour', 'holiday','workingday',
                                                     'season','weather'])
```

(8) LinearRegression, Ridge, Lasso 학습 및 예측 성능 확인

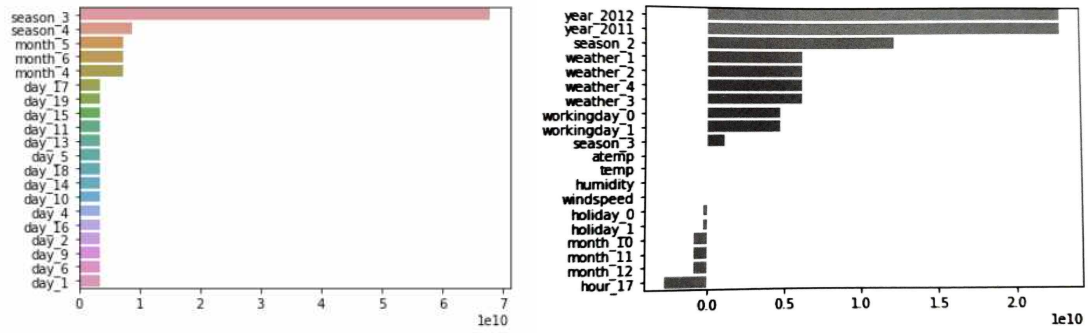
```
# 원-핫 인코딩이 적용된 feature 데이터 세트 기반으로 학습/예측 데이터 분할.
X_train, X_test, y_train, y_test = train_test_split(X_features_ohe, y_target_log,
                                                    test_size=0.3, random_state=0)

# 모델과 학습/테스트 데이터 셋을 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

# model 별로 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: 97.686, MAE: 63.381
### Ridge ###
RMSLE: 0.590, RMSE: 98.529, MAE: 63.893
### Lasso ###
RMSLE: 0.635, RMSE: 113.219, MAE: 72.803
```

(9) 회귀 계수 상위 피처 추출

```
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)
```

```
<AxesSubplot:>
```



(파마완 그래프)

1.6 회귀 트리

* 랜덤 포레스트, GBM, XGBoost, LightGBM 이용해 성능 평가

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

# 랜덤 포레스트, GBM, XGBoost, LightGBM model 별로 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 입력 될 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_expml=True)
```

```
### RandomForestRegressor ###
RMSLE: 0.355, RMSE: 50.247, MAE: 31.054
### GradientBoostingRegressor ###
RMSLE: 0.330, RMSE: 53.327, MAE: 32.739
### XGBRegressor ###
RMSLE: 0.342, RMSE: 51.732, MAE: 31.251
### LGBMRegressor ###
RMSLE: 0.319, RMSE: 47.215, MAE: 29.029
```

02 주택 가격 예측

EWHA EURON

2.1 대회 소개

- 79개의 설명 변수가 미국 아이오와 주의 에임스에 있는 주거용 주택의 거의 모든 측면을 설명함.
- 대회는 이를 이용해 각 주택의 최종 가격을 예측하는 데 도전함.

※ 분석 방법

- 회귀 분석

※ 성능 평가

- RMSLE 이용
  (가격이 비싼 주택일수록 예측 결과 오류가 전체 오류에 미치는 비중이 높으므로 이것을 상쇄하기 위해 오류 값을 로그 변환한 RMSLE를 이용함.)

2.2 Data Description

1. SalePrice : 부동산의 판매 가격(단위 : 달러)
2. GrLivArea : 주거 공간 크기
3. CentralAir : 중앙 에어컨
4. OverallQual : 전반적인 재료 및 마무리 품질
5. OverallCond : 전체 상태 등급
6. RoofStyle : 지붕의 유형
7. 1stFirSF : 1층 평방 피트
8. PaveDrive : 포장된 진입로
9. Fence : 울타리 품질
10. Sale Type : 판매 유형
11. LotFrontage : 재산에 연결된 거리의 선형 피트
12. Street : 도로 접근의 유형

2.2 Data Description

(1) 데이터 로드 및 확인

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

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl

3 rows × 81 columns

(2) Null 칼럼 및 건수

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

```
데이터 세트의 Shape: (1460, 81)

전체 feature 들의 type
 object     43
int64       35
float64      3
dtype: int64

Null 컬럼과 그 건수:
  PoolQC          1453
MiscFeature       1406
Alley             1369
Fence             1179
FireplaceQu        690
LotFrontage        259
GarageYrBlt         81
GarageType          81
GarageFinish        81
GarageQual          81
GarageCond          81
BsmtFinType2        38
BsmtExposure        38
BsmtFinType1        37
BsmtCond            37
BsmtQual            37
MasVnrArea           8
MasVnrType           8
Electrical           1
dtype: int64
```

Null 값이 너무 많은 피처는 드롭

EWHA EURON

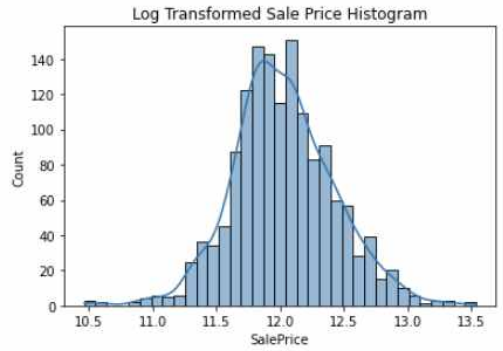
2.3 Data Processing

(3) 타깃 값의 분포도 확인

```
plt.title('Original Sale Price Histogram')
plt.xticks(rotation=15)
sns.histplot(house_df['SalePrice'], kde=True)
plt.show()
```

(4) 타깃 값의 로그 변환 후 분포도 확인

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



- 정규 분포 형태로 변환하기 위해 로그 변환 적용
- 넘파이의 log1p( )를 이용해 로그 변환한 결괏값을 기반으로 학습한 뒤, 예측시 다시 결괏값을 expm1( )으로 추후에 환원하면 됨.

(5) 로그 변환 및 Null 값 처리

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

# Null 이 너무 많은 컬럼들과 불필요한 컬럼 삭제
house_df.drop(['Id','PoolQC' , 'MiscFeature', 'Alley', 'Fence','FireplaceQu'],
              axis=1 , inplace=True)
# Drop 하지 않는 숫자형 Null컬럼들은 평균값으로 대체
house_df.fillna(house_df.mean(),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
```

(6) one-hot-encoing

```
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, 271)
## Null 피처의 Type :
 Series([], dtype: object)
```

2.4 선형 회귀 모델 학습/예측/평가

(1) RMSE 측정 함수 생성

```python
def get_rmse(model):
    pred = model.predict(X_test)
    mse = mean_squared_error(y_test , pred)
    rmse = np.sqrt(mse)
    print('{0} 로그 변환된 RMSE: {1}'.format(model.__class__.__name__,np.round(rmse, 3)))
    return rmse

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

(2) 선형 회귀 모델 학습/예측/평가

```python
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: 0.128
Lasso 로그 변환된 RMSE: 0.176
```

라쏘의 경우 최적 하이퍼 파라미터 튜닝 필요.

```
[0.1318957657915412, 0.12750846334053026, 0.17628250556471398]
```

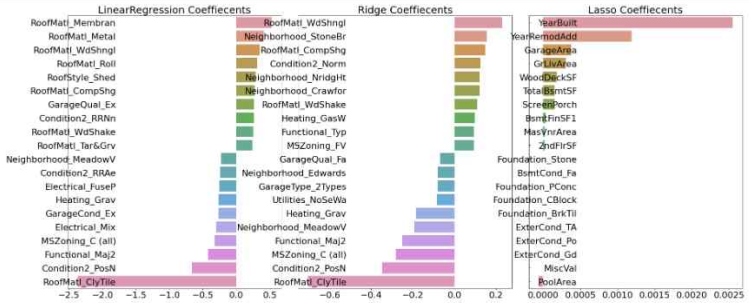
(3) 상위 10개, 하위 10개 피처명과 회귀 계수

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

    # + 상위 10개 , - 하위 10개 coefficient 추출하여 반환.
    coef_high = coef.sort_values(ascending=False).head(10)
    coef_low = coef.sort_values(ascending=False).tail(10)
    return coef_high, coef_low
```

(4) 모델별 회귀 계수 시각화

```python
def visualize_coefficient(models):
    # 3개 회귀 모델의 시각화를 위해 3개의 컬럼을 가지는 subplot 생성
    fig, axs = plt.subplots(figsize=(24,10),nrows=1, ncols=3)
    fig.tight_layout()
    # 입력인자로 받은 list객체인 models에서 차례로 model을 추출하여 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] )
# 순차적으로 ax subplot에 barchar로 표현, 한 화면에 표현하기 위해 tick label 위치와 font 크기 조정.
        axs[i_num].set_title(model.__class__.__name__+' Coeffiecents', 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)
```

라쏘의 경우 두 개의 모델과 다른 회귀 계수 형태를 보임.



2.4 선형 회귀 모델 학습/예측/평가

(5) 5개의 교차 검증 폴드 세트로 불할해 평균 RSME 측정

```
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 RMSE값과 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 lr_reg, ridge_reg, lasso_reg 모델의 CV RMSE값 출력
models = [lr_reg, ridge_reg, lasso_reg]
get_avg_rmse_cv(models)
```

```
LinearRegression CV RMSE 값 리스트: [1.350000e-01 1.650000e-01 1.680000e-01 1.494232e+03 1.980000e-01]
LinearRegression CV 평균 RMSE 값: 298.979

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

라쏘의 경우 릿지 모델보다 성능 떨어짐.

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

(6) Alpha 하이퍼 파라미터 변화시키면서 최적 값 도출

```
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_))
    return grid_model.best_estimator_

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] }
best_rige = print_best_params(ridge_reg, ridge_params)
best_lasso = 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}
```

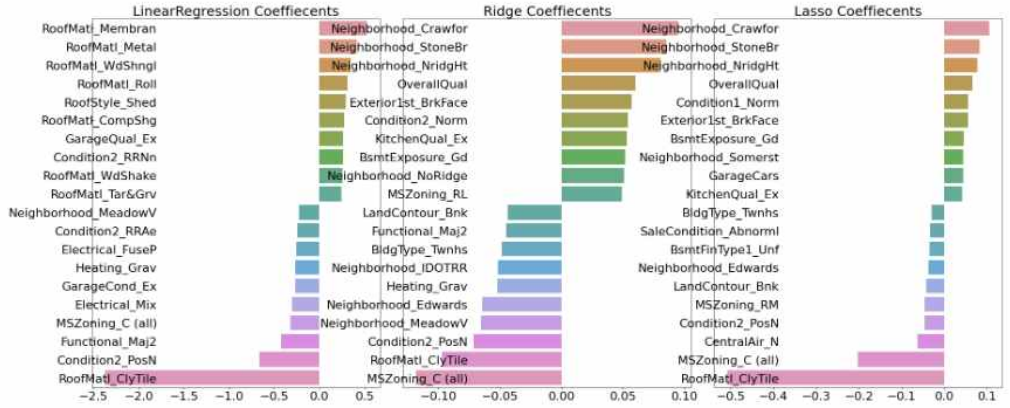
라쏘의 경우 성능 좋아짐

(7) 모델의 학습/예측/평가 수행 및 회귀 계수 시각화

```
# 앞의 최적화 alpha값으로 학습데이터로 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)
```

- 릿지와 라쏘 모델에서 비슷한 피처의 회귀 계수가 높음.
- 라쏘 모델의 경우는 릿지에 비해 동일한 피처라도 회귀 계수 값이 상당히 작음.

```
LinearRegression 로그 변환된 RMSE: 0.132
Ridge 로그 변환된 RMSE: 0.124
Lasso 로그 변환된 RMSE: 0.12
```



2.4 선형 회귀 모델 학습/예측/평가

※ 피처 데이터의 경우도 지나치게 왜곡된 피처가 존재할 경우 회귀 예측 성능을 저하시킬 수 있음

- 사이파이 stats 모듈의 skew( ) 함수를 이용해 칼럼의 데이터 세트의 왜곡된 정도를 쉽게 추출할 수 있음.

- 일반적으로 skew( )함수의 반환 값이 1 이상인 경우를 왜곡 정도가 높다고 판단하지만, 상황에 따라 편차 있음.

- skew( )를 적용하는 숫자형 피처에서 원-핫 인코딩 된 카테고리 숫자형 피처는 제외해야 함.
  (카테고리 피처는 코드성 피처이므로 인코딩 시 당연히 왜곡될 가능성이 높기 때문)

2.4 선형 회귀 모델 학습/예측/평가

(8) skew( ) 함수 적용

```
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
MSSubClass        1.406210
1stFlrSF          1.375342
GrLivArea         1.365156
dtype: float64
```

(9) 왜곡 정도가 높은 피처 로그 변환

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

(10) one-hot encoding & 피처/타깃 데이터 세트 생성 & 최적 alpha값과 RMSE 출력

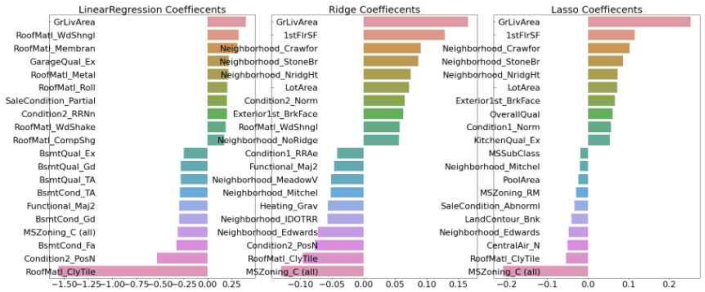
```
# 왜곡 정도가 높은 피처들을 로그 변환 했으므로 다시 원-핫 인코딩 적용 및 피처/타겟 데이터 셋 생성
house_df_ohe = pd.get_dummies(house_df)
y_target = house_df_ohe['SalePrice']
X_features = house_df_ohe.drop('SalePrice',axis=1, inplace=False)
X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2,
                                                    random_state=156)

# 피처들을 로그 변환 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] }
best_ridge = print_best_params(ridge_reg, ridge_params)
best_lasso = 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}
```

(11) 모델의 학습/예측/평가 및 모델별 회귀 계수 시각화

```
# 앞의 최적화 alpha값으로 학습데이터로 학습, 테스트 데이터로 예측 및 평가 수행.
lr_reg = LinearRegression()
lr_reg.fit(X_train, y_train)
ridge_reg = Ridge(alpha=10)
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)
```

```
LinearRegression 로그 변환된 RMSE: 0.128
Ridge 로그 변환된 RMSE: 0.122
Lasso 로그 변환된 RMSE: 0.119
```



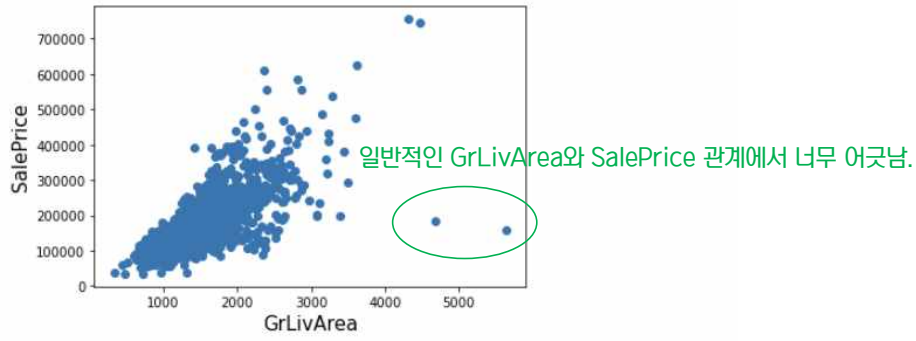
2.4 선형 회귀 모델 학습/예측/평가

※ Outlier 데이터

- 회귀 계수가 높은 피처, 즉 예측에 많은 영향을 미치는 중요 피처의 이상치 데이터의 처리 중요함.

(12) GrLivArea와 SalePrice의 관계 시각화

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



일반적인 GrLivArea와 SalePrice 관계에서 너무 어긋남.

(13) Outlier 삭제

```
# GrLivArea와 SalePrice 모두 로그 변환되었으므로 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('아웃라이어 레코드 index :', outlier_index.values)
print('아웃라이어 삭제 전 house_df_ohe shape:', house_df_ohe.shape)
# DataFrame의 index를 이용하여 아웃라이어 레코드 삭제.
house_df_ohe.drop(outlier_index , axis=0, inplace=True)
print('아웃라이어 삭제 후 house_df_ohe shape:', house_df_ohe.shape)
```

```
아웃라이어 레코드 index : [ 523 1298]
아웃라이어 삭제 전 house_df_ohe shape: (1460, 271)
아웃라이어 삭제 후 house_df_ohe shape: (1458, 271)
```

2.4 선형 회귀 모델 학습/예측/평가

(14) 릿지와 라쏘 모델의 최적화 수행

```
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] }
best_ridge = print_best_params(ridge_reg, ridge_params)
best_lasso = 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}
```

예측 수치 매우 크게 향상

- 보통 머신러닝 프로세스 중에서 데이터의 가공은 알고리즘을 적용하기 이전에 수행하지만, 완벽하게 데이터의 선처리 작업을 수행하라는 의미는 아님.
- 대략의 데이터 가공과 모델 최적화를 수행한 뒤 다시 이에 기반한 여러 가지 기법의 데이터 가공과 하이퍼 파라미터 기반의 모델 최적화를 반복적으로 수행하는 것이 바람직한 머신러닝 모델 생성 과정.

(15) RMSE 수치 및 회귀 계수를 시각화

```
# 앞의 최적화 alpha값으로 학습데이터로 학습, 테스트 데이터로 예측 및 평가 수행.
lr_reg = LinearRegression()
lr_reg.fit(X_train, y_train)
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)

# 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)
```

```
LinearRegression 로그 변환된 RMSE: 0.129
Ridge 로그 변환된 RMSE: 0.103
Lasso 로그 변환된 RMSE: 0.1
```

EWHA EURON

2.5 회귀 트리 모델 학습/예측/평가

(1) XGBoost - 5폴드 세트에 대한 최적 평균 RMSE 값

```python
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)
best_xgb = print_best_params(xgb_reg, xgb_params)
```

```
XGBRegressor 5 CV 시 최적 평균 RMSE 값: 0.1178, 최적 alpha:{'n_estimators': 1000}
```

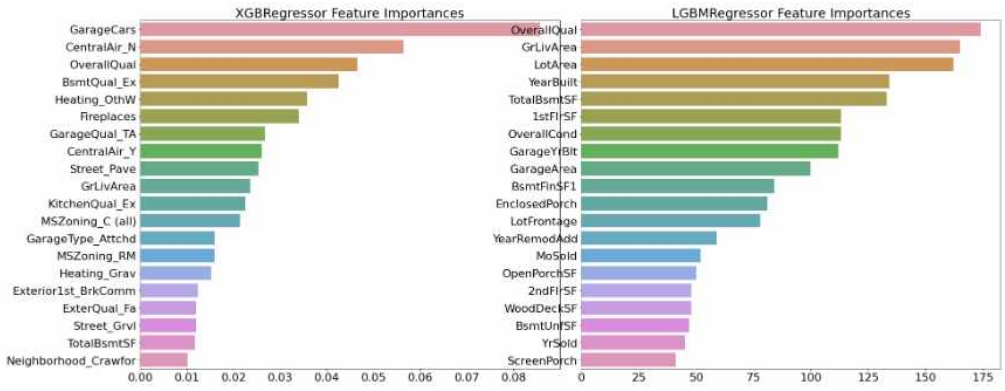
(2) LightGBM – 5폴드 세트에 대한 최적 평균 RMSE 값

```python
from lightgbm import LGBMRegressor

lgbm_params = {'n_estimators':[1000]}
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)
best_lgbm = print_best_params(lgbm_reg, lgbm_params)
```

```
LGBMRegressor 5 CV 시 최적 평균 RMSE 값: 0.1163, 최적 alpha:{'n_estimators': 1000}
```

(3) XGBoost, LightGBM 피처 중요도 시각화

```python
# 모델의 중요도 상위 20개의 피처명과 그때의 중요도값을 Series로 반환.
def get_top_features(model):
    ftr_importances_values = model.feature_importances_
    ftr_importances = pd.Series(ftr_importances_values, index=X_features.columns  )
    ftr_top20 = ftr_importances.sort_values(ascending=False)[:20]
    return ftr_top20

def visualize_ftr_importances(models):
    # 2개 회귀 모델의 시각화를 위해 2개의 컬럼을 가지는 subplot 생성
    fig, axs = plt.subplots(figsize=(24,10),nrows=1, ncols=2)
    fig.tight_layout()
    # 입력인자로 받은 list객체인 models에서 차례로 model을 추출하여 피처 중요도 시각화.
    for i_num, model in enumerate(models):
        # 중요도 상위 20개의 피처명과 그때의 중요도값 추출
        ftr_top20 = get_top_features(model)
        axs[i_num].set_title(model.__class__.__name__+' Feature Importances', size=25)
        #font 크기 조정.
        for label in (axs[i_num].get_xticklabels() + axs[i_num].get_yticklabels()):
            label.set_fontsize(22)
        sns.barplot(x=ftr_top20.values, y=ftr_top20.index , ax=axs[i_num])

models = [best_xgb, best_lgbm]
visualize_ftr_importances(models)
```



2.6 회귀 모델의 예측 결과 혼합을 통한 최종 예측

※ 예측 결과 혼합

- A모델과 B 모델, 두 모델의 예측 값이 있다면A 모델 예측 값의 40%, B 모델 예측 값의 60%를 더해서 최종 회귀 값으로 예측하는 것.

Ex)

A 회귀 모델 예측값 : [100, 80, 60]

B 회귀 모델 예측값 : [120, 80, 50]

최종 회귀 예측값 : [100*0.4 + 120*0.6, 80*0.4 + 80*0.6, 60*0.4 + 50*0.6] = [112, 80, 54]

#2.6 회귀 모델의 예측 결과 혼합을 통한 최종 예측

(1) 릿지 모델과 라쏘 모델 혼합

```python
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, 개별모델의 RMSE 값 출력
get_rmse_pred(preds)
```

```
최종 혼합 모델의 RMSE: 0.10007930884470488
Ridge 모델의 RMSE: 0.10345177546603221
Lasso 모델의 RMSE: 0.10024170460890013
```

(2) XGBoost와 LightGBM을 혼합

```python
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)
```

```
최종 혼합 모델의 RMSE: 0.10170077353447762
XGBM 모델의 RMSE: 0.10738295638346222
LGBM 모델의 RMSE: 0.10382510019327311
```

최종 혼합 모델의 RMSE가 개별 모델보다 성능 면에서 약간 개선됨.

EWHA EURON

2.7 스태킹 앙상블 모델을 통한 회귀 예측

※ 스태킹 모델의 구현 방법

- 스태킹 모델은 두 종류의 모델이 필요함
  1) 개별적인 기반 모델
  2) 1)의 개별 기반 모델의 예측 데이터를 학습 데이터로 만들어서 학습하는 최종 메타 모델

- 스태킹 모델의 핵심은 여러 개별 모델의 예측 데이터를 각각 스태킹 형태로 결합해 최종 메타 모델의 학습용 피처 데이터 세트와 테스트용 피처 데이터 세트를 만드는 것.

- 최종 메타 모델이 학습할 피처 데이터 세트는 원복 학습 피처 세트로 학습한 개별 모델의 예측 값을 스태킹 형태로 결합한 것.

2.7 스태킹 앙상블 모델을 통한 회귀 예측

(1)데이터 세트 생성 함수

```python
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_folds값으로 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 셋 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 세트내 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
```

(2) 모델별로 적용해 메타 모델이 사용할 학습/피처 데이터 세트 추출

```python
# get_stacking_base_datasets( )은 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
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
Lasso  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
XGBRegressor  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
LGBMRegressor  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
```

2.7 스태킹 앙상블 모델을 통한 회귀 예측

(3) 최종 메타 모델에 적용(라쏘 모델 이용) & 예측 및 RMSE 측정

```
# 개별 모델이 반환한 학습 및 테스트용 데이터 세트를 Stacking 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)

#기반 모델의 예측값을 기반으로 새롭게 만들어진 학습 및 테스트용 데이터로 예측하고 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: 0.0979915296518969
```

- 스태킹 회귀 모델을 적용한 결과, 테스트 데이터 세트에서 RMSE가 약 0.0979로 현재까지 가장 좋은 성능 평가를 보여줌.
- 스태킹 모델은 분류뿐만 아니라 회귀에서 특히 효과적으로 사용될 수 있는 모델.

EWHA EURON

03.AutoML

EWHA EURON

3.1 AutoML

AutoML(Automated Machine Learning)
- 현재의 머신러닝 모델링은 Machine Learning Process 동안 많은 시간과 노력이 요구됨
- AutoML은 기계 학습 파이프 라인에서 수작업과 반복되는 작업을 자동화하는 프로세스
- 머신러닝을 자동화하는 AI 기술

*Machine Learning Process : 문제 정의 과정, 데이터 수집, 전처리, 모델 학습 및 평가, 서비스 적용
*파이프 라인 : 한 데이터 처리 단계의 출력이 다음 단계의 입력으로 이어지는 형태로 연결된 구조

AutoML systems
- AutoWEKA
- Auto-sklearn
- Auto-PyTorch

3.2 Pycaret

PyCaret
- Low-code machine learning
- AutoML을 하게 해주는 파이썬 라이브러리
- scikit-learn 패키지 기반, 분류, 회귀, 군집화 등 다양한 모델 지원

*Low-code : 어플리케이션과 시스템을 빌딩할 때 거의 코딩이 필요 없는 방식의 소프트웨어

04.PyCaret을 활용한 주택 가격 예측

EWHA EURON

4.1 Data Description

Data Import

```
import pandas as pd
train = pd.read_csv('/content/train.csv')
test = pd.read_csv('/content/test.csv')
train.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	BldgType	HouseStyle
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Inside	Gtl	CollgCr	Norm	Norm	1Fam	2Story
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	FR2	Gtl	Veenker	Feedr	Norm	1Fam	1Story
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Inside	Gtl	CollgCr	Norm	Norm	1Fam	2Story
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	Corner	Gtl	Crawfor	Norm	Norm	1Fam	2Story
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	FR2	Gtl	NoRidge	Norm	Norm	1Fam	2Story

4.2 Install PyCaret

```
!pip install pycaret
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting pycaret
  Using cached pycaret-2.3.10-py3-none-any.whl (320 kB)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from pycaret) (3.5.3)
Requirement already satisfied: Boruta in /usr/local/lib/python3.7/dist-packages (from pycaret) (0.3)
Requirement already satisfied: imbalanced-learn==0.7.0 in /usr/local/lib/python3.7/dist-packages (from pycaret) (0.7.0)
Requirement already satisfied: lightgbm>=2.3.1 in /usr/local/lib/python3.7/dist-packages (from pycaret) (3.3.2)
Requirement already satisfied: yellowbrick>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from pycaret) (1.3.post1)
Requirement already satisfied: kmodes>=0.10.1 in /usr/local/lib/python3.7/dist-packages (from pycaret) (0.12.2)
Requirement already satisfied: scikit-plot in /usr/local/lib/python3.7/dist-packages (from pycaret) (0.3.7)
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (from pycaret) (1.3.5)
```

```
from pycaret.utils import enable_colab
enable_colab()
```

```
Colab mode enabled.
```

*!pip install pycaret[full] : 필요한 모델이 다 설치되지 않았을 때는 pycaret[full]로 코드 돌리기
*pycaret의 버전과 scikit-learn의 버전, xgboost의 버전 등등 서로 필요한 버전이 달라 오류가 발생할 수 있음
-> 오류 창에 뜨는 문구를 통해 버전을 upgrad하거나 downgrade 시켜주기
(저의 경우, scikit-learn은 0.23.2로 downgrade, xgboost는 1.1.0으로 upgrade 시켜주었습니다!

4.3 Flow

Iteration 마다

setup

Compare models

Create and Store Models in Variable

Blend Models

Stack Models

4.4 Iteration1: Setup without Preprocessing

```
from pycaret.regression import *
reg1 = setup(train, target = 'SalePrice', session_id = 123, silent = True)
#silent is set to True for unattended run during kernel execution
```

	Description	Value
0	session_id	123
1	Target	SalePrice
2	Original Data	(1460, 81)
3	Missing Values	True
4	Numeric Features	19
5	Categorical Features	61
6	Ordinal Features	False
7	High Cardinality Features	False
8	High Cardinality Method	None
9	Transformed Train Set	(1021, 405)
10	Transformed Test Set	(439, 405)
11	Shuffle Train-Test	True
12	Stratify Train-Test	False
13	Fold Generator	KFold
14	Fold Number	10
15	CPU Jobs	-1
16	Use GPU	False
17	Log Experiment	False

setup( )
session_id
: Random seed를 설정해주는 부분
즉 난수 생성에 측정한 seed 값을 적용하여 Random하게 일을 처리,
이후에 반복 실행을 했을 때에도, 동일한 결과가 나올 수 있도록 함

silent =True
: setup시 중간에 피쳐 속성을 확인하고 엔터를 쳐주어야 하는데
알아서 넘아가게 해줌

https://pycaret.readthedocs.io/en/stable/api/regression.html
https://sthsb.tistory.com/31

EWHA EURON

4.4 Iteration1: Setup without Preprocessing

models( )
: 모델 확인, setup을 한 후에만 가능

```
models()
```

```
INFO:logs:gpu_param set to False
```

ID	Name	Reference	Turbo
lr	Linear Regression	sklearn.linear_model._base.LinearRegression	True
lasso	Lasso Regression	sklearn.linear_model._coordinate_descent.Lasso	True
ridge	Ridge Regression	sklearn.linear_model._ridge.Ridge	True
en	Elastic Net	sklearn.linear_model._coordinate_descent.Elast...	True
lar	Least Angle Regression	sklearn.linear_model._least_angle.Lars	True
llar	Lasso Least Angle Regression	sklearn.linear_model._least_angle.LassoLars	True
omp	Orthogonal Matching Pursuit	sklearn.linear_model._omp.OrthogonalMatchingPu...	True
br	Bayesian Ridge	sklearn.linear_model._bayes.BayesianRidge	True
ard	Automatic Relevance Determination	sklearn.linear_model._bayes.ARDRegression	False
par	Passive Aggressive Regressor	sklearn.linear_model._passive_aggressive.Passi...	True
ransac	Random Sample Consensus	sklearn.linear_model._ransac.RANSACRegressor	False
tr	TheilSen Regressor	sklearn.linear_model._theil_sen.TheilSenRegressor	False
huber	Huber Regressor	sklearn.linear_model._huber.HuberRegressor	True
kr	Kernel Ridge	sklearn.kernel_ridge.KernelRidge	False
svm	Support Vector Regression	sklearn.svm._classes.SVR	False
knn	K Neighbors Regressor	sklearn.neighbors._regression.KNeighborsRegressor	True
dt	Decision Tree Regressor	sklearn.tree._classes.DecisionTreeRegressor	True
rf	Random Forest Regressor	sklearn.ensemble._forest.RandomForestRegressor	True
et	Extra Trees Regressor	sklearn.ensemble._forest.ExtraTreesRegressor	True
ada	AdaBoost Regressor	sklearn.ensemble._weight_boosting.AdaBoostRegr...	True
gbr	Gradient Boosting Regressor	sklearn.ensemble._gb.GradientBoostingRegressor	True
mlp	MLP Regressor	sklearn.neural_network._multilayer_perceptron....	False
xgboost	Extreme Gradient Boosting	xgboost.sklearn.XGBRegressor	True
lightgbm	Light Gradient Boosting Machine	lightgbm.sklearn.LGBMRegressor	True
catboost	CatBoost Regressor	catboost.core.CatBoostRegressor	True
dummy	Dummy Regressor	sklearn.dummy.DummyRegressor	True

EWHA EURON

4.4 Iteration1: Setup without Preprocessing

compare models
: models()에서 제공하는 모델들이나 scikit-learn에서 제공하는 모델을 별도로 선언한 이후에 입력한 모델들의 성능(MAE, MSE, RMSE,R^2, Train Time)등 DataFrame의 형태로 제공

```
compare_models()
```

	Model	MAE	MSE	RMSE	R2	RMSLE	MAPE	TT (Sec)
catboost	CatBoost Regressor	1.619247e+04	8.104013e+08	2.768092e+04	8.811000e-01	0.1327	0.0945	8.614
gbr	Gradient Boosting Regressor	1.818001e+04	8.748905e+08	2.902744e+04	8.692000e-01	0.1435	0.1064	0.747
lightgbm	Light Gradient Boosting Machine	1.831749e+04	9.469400e+08	3.006597e+04	8.603000e-01	0.1510	0.1086	0.357
xgboost	Extreme Gradient Boosting	1.887950e+04	1.021791e+09	3.089199e+04	8.497000e-01	0.1514	0.1098	1.989
rf	Random Forest Regressor	1.953262e+04	1.101270e+09	3.233810e+04	8.382000e-01	0.1593	0.1158	2.523
ada	AdaBoost Regressor	2.684982e+04	1.404570e+09	3.714260e+04	7.852000e-01	0.2130	0.1752	0.766
ridge	Ridge Regression	2.048344e+04	1.506927e+09	3.665019e+04	7.705000e-01	0.1955	0.1242	0.060
omp	Orthogonal Matching Pursuit	1.876735e+04	1.480648e+09	3.566383e+04	7.703000e-01	0.1656	0.1108	0.038
et	Extra Trees Regressor	2.289222e+04	1.621210e+09	3.928796e+04	7.565000e-01	0.1853	0.1333	2.737
en	Elastic Net	2.161149e+04	1.710967e+09	3.855268e+04	7.477000e-01	0.1723	0.1269	0.207
lasso	Lasso Regression	2.065010e+04	1.660499e+09	3.852132e+04	7.390000e-01	0.2032	0.1261	0.189
br	Bayesian Ridge	2.572986e+04	2.118898e+09	4.356422e+04	6.838000e-01	0.2080	0.1514	0.263
dt	Decision Tree Regressor	2.957314e+04	2.265758e+09	4.670944e+04	6.618000e-01	0.2321	0.1710	0.067
knn	K Neighbors Regressor	3.001398e+04	2.244784e+09	4.660246e+04	6.606000e-01	0.2275	0.1766	0.101
lr	Linear Regression	2.422727e+04	2.170662e+09	4.439976e+04	6.560000e-01	0.2504	0.1535	1.013
huber	Huber Regressor	2.859871e+04	2.287949e+09	4.586925e+04	6.518000e-01	0.2323	0.1745	0.717
dummy	Dummy Regressor	5.821813e+04	6.558664e+09	8.044672e+04	-6.500000e-03	0.4146	0.3714	0.027

4.4 Iteration1: Setup without Preprocessing

Create and Store Models in Variable

create_model( )

: 여러 모델이 아닌 하나의 모델에 대해서 setup()의 설정대로 학습을 진행하고 학습의 결과를 확인 가능
또한 세부적으로 각 fold에 대한 성능을 제시함

Verbose

: 함수 수행 시 발생하는 상세한 정보들을 표준 출력으로 자세히 내보낼 것인가
True의 경우 자세히 출력

```
catboost = create_model('catboost', verbose = False)
#verbose set to False to avoid printing score grid
gbr = create_model('gbr', verbose = False)
xgboost = create_model('xgboost', verbose = False)
```

4.4 Iteration1: Setup without Preprocessing

Blend Models
blend_models
: 선택한 모델들을 혼합한 모델을 생성

```
blend_top_3 = blend_models(estimator_list = [catboost, gbr, xgboost])
```

Fold	MAE	MSE	RMSE	R2	RMSLE	MAPE
0	19387.5230	1.649128e+09	40609.4518	0.7986	0.1734	0.1105
1	17398.6943	9.211269e+08	30350.0724	0.8563	0.1293	0.0913
2	13022.4087	3.492597e+08	18688.4911	0.9244	0.1212	0.0863
3	14998.9957	6.210488e+08	24920.8507	0.8803	0.1269	0.0895
4	19240.1068	1.484830e+09	38533.4961	0.8490	0.1740	0.1194
5	16780.1661	7.337058e+08	27087.0045	0.8984	0.1337	0.0993
6	15704.9596	5.045826e+08	22462.9172	0.8930	0.1103	0.0879
7	17991.8100	7.652387e+08	27662.9476	0.8877	0.1439	0.1048
8	17192.1957	6.653228e+08	25793.8527	0.9007	0.1289	0.0975
9	14928.5385	5.540796e+08	23538.8955	0.8996	0.1090	0.0803
Mean	16664.5398	8.248323e+08	27964.7980	0.8788	0.1351	0.0967
Std	1913.4851	4.006489e+08	6542.3514	0.0339	0.0217	0.0115

4.4 Iteration1: Setup without Preprocessing

Stack Models

```
stack1 = stack_models(estimator_list = [gbr, xgboost], meta_model = catboost, restack = True)
```

Fold	MAE	MSE	RMSE	R2	RMSLE	MAPE
0	19942.3701	1.746750e+09	41794.1422	0.7866	0.1783	0.1151
1	16031.0017	7.951136e+08	28197.7593	0.8759	0.1258	0.0883
2	12634.7944	3.232107e+08	17978.0603	0.9300	0.1180	0.0836
3	14803.3846	5.928873e+08	24349.2782	0.8857	0.1265	0.0890
4	18769.2597	1.322751e+09	36369.6493	0.8654	0.1644	0.1131
5	17659.9283	1.163600e+09	34111.5840	0.8388	0.1307	0.0974
6	14949.2886	5.449366e+08	23343.8769	0.8844	0.1091	0.0816
7	16631.4183	6.283064e+08	25066.0414	0.9078	0.1304	0.0949
8	16867.6848	6.474462e+08	25444.9633	0.9034	0.1310	0.0974
9	14699.5851	5.248870e+08	22910.4117	0.9049	0.1144	0.0821
Mean	16298.8716	8.289890e+08	27956.5767	0.8783	0.1329	0.0943
Std	2043.9496	4.191837e+08	6886.1300	0.0389	0.0207	0.0114

https://koreapy.tistory.com/744

4.4 Iteration1: Setup without Preprocessing

Stack Models
stack_models()
: 메타 학습을 사용하는 앙상블 방법으로 stacking ensemble 방법을 구현한 모델
estimator_list
: 매개변수를 사용하여 훈련된 모델 목록 가져옴
  compare_model( )에서 성능이 잘 나온 모델들을
  선택하는 파라미터를 적용시켜서 사용할 수 있음
meta_model
: 메타 모델의 입력으로 사용됨
Restack
: 원시 데이터를 메타 모델에 노출하는 기능을 제어
  기본적으로 True,
  False의 경우 메타 모델은
  기본 모델의 예측만 사용하여 최종 예측 생성

*meta model – 실제 모델을 대체할 수 있는 근사 모델
*PyCaret 2.x의 향후 릴리즈에서 더 이상 사용되지 않음

https://koreapy.tistory.com/744

4.5 Iteration2: Setup with Preprocessing

```
from pycaret.regression import *
reg1 = setup(train, target = 'SalePrice', session_id = 123,
             normalize = True, normalize_method = 'zscore',
             transformation = True, transformation_method = 'yeo-johnson', transform_target = True,
             ignore_low_variance = True, combine_rare_levels = True,
             numeric_features=['OverallQual', 'OverallCond', 'BsmtFullBath', 'BsmtHalfBath',
                               'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr',
                               'TotRmsAbvGrd', 'Fireplaces', 'GarageCars', 'PoolArea'],
             silent = True #silent is set to True for unattended run during kernel execution
             )
```

	Description	Value
0	session_id	123
1	Target	SalePrice
2	Original Data	(1460, 81)
3	Missing Values	True
4	Numeric Features	31
5	Categorical Features	49
6	Ordinal Features	False
7	High Cardinality Features	False
8	High Cardinality Method	None
9	Transformed Train Set	(1021, 242)
10	Transformed Test Set	(439, 242)

4.5 Iteration2: Setup with Preprocessing

normalize
: 데이터에 정규화를 할 것인지
normalize_method
: normalize = True인 경우, 어떤 방식으로 정규화를 진행할 것인지 설정
transformation
: power transformation을 통해 데이터 샘플들의 분포가
가우시안 분포(정규 분포)에 더 가까워지도록 처리해주는 과정

https://dsbook.tistory.com/360
https://pycaret.gitbook.io/docs/get-started/preprocessing/scale-and-transform

4.6 Iteration3: Setup with Advance Preprocessing

```
from pycaret.regression import *
reg1 = setup(train, target = 'SalePrice', session_id = 123,
             normalize = True, normalize_method = 'zscore',
             transformation = True, transformation_method = 'yeo-johnson', transform_target = True,
             numeric_features=['OverallQual', 'OverallCond', 'BsmtFullBath', 'BsmtHalfBath',
                               'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr',
                               'TotRmsAbvGrd', 'Fireplaces', 'GarageCars', 'PoolArea'],
             ordinal_features= {'ExterQual': ['Fa', 'TA', 'Gd', 'Ex'],
                                'ExterCond' : ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                'BsmtQual' : ['Fa', 'TA', 'Gd', 'Ex'],
                                'BsmtCond' : ['Po', 'Fa', 'TA', 'Gd'],
                                'BsmtExposure' : ['No', 'Mn', 'Av', 'Gd'],
                                'HeatingQC' : ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                'KitchenQual' : ['Fa', 'TA', 'Gd', 'Ex'],
                                'FireplaceQu' : ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                'GarageQual' : ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                'GarageCond' : ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                'PoolQC' : ['Fa', 'Gd', 'Ex']},
             polynomial_features = True, trigonometry_features = True, remove_outliers = True, outliers_threshold = 0.01,
             silent = True #silent is set to True for unattended run during kernel execution
             )
```

	Description	Value
0	session_id	123
1	Target	SalePrice
2	Original Data	(1460, 81)
3	Missing Values	True
4	Numeric Features	31

4.7 Tune Models

create_model
: 특정 ML을 만들고 학습시킴
tune_model( )
: 입력한 모델에 대해서 hyper-parameter tuning을 수행
n_iter
: tuning하여 성능을 비교할 후보군의 수
  이 값을 크게 해줄수록 성능이 더 좋아질 가능성은 높지만
  더 많은 시간이 걸리므로 각 task에 맞게 적절한 값을 설정해야 함

4.7 Tune Models

```
huber = create_model('huber', max_iter = 100)
```

Fold	MAE	MSE	RMSE	R2	RMSLE	MAPE
0	19552.9764	2.994087e+09	54718.2532	0.6269	0.1890	0.1142
1	14980.0067	5.122814e+08	22633.6350	0.9228	0.1127	0.0804
2	13681.5826	4.024048e+08	20060.0298	0.9130	0.1190	0.0855
3	14736.2822	4.803659e+08	21917.2504	0.9047	0.1088	0.0846
4	14028.6435	4.959637e+08	22270.2415	0.9503	0.1442	0.0957
5	14558.8642	6.712590e+08	25908.6666	0.9077	0.1128	0.0822
6	12958.1242	3.238576e+08	17996.0444	0.9241	0.0943	0.0743
7	15099.1037	5.720963e+08	23918.5354	0.9198	0.1221	0.0820
8	13417.5207	3.775311e+08	19430.1594	0.9433	0.1063	0.0786
9	13799.6079	4.551584e+08	21334.4426	0.9180	0.1176	0.0811
Mean	14681.2712	7.285005e+08	25018.7258	0.8931	0.1227	0.0859
Std	1754.5450	7.609853e+08	10127.3839	0.0898	0.0252	0.0108

```
huber = tune_model(huber)
```

Fold	MAE	MSE	RMSE	R2	RMSLE	MAPE
0	18871.7826	2.672425e+09	51695.5072	0.6670	0.1858	0.1111
1	14747.8708	5.353453e+08	23137.5291	0.9194	0.1142	0.0806
2	13427.9696	3.999654e+08	19999.1344	0.9136	0.1182	0.0841
3	14637.2492	4.991847e+08	22342.4413	0.9010	0.1094	0.0835
4	14983.5547	6.100456e+08	24699.1009	0.9388	0.1485	0.0995
5	14666.7591	6.640395e+08	25768.9633	0.9087	0.1134	0.0822
6	12744.9777	3.234912e+08	17985.8601	0.9241	0.0948	0.0738
7	14608.2155	5.219037e+08	22845.2120	0.9268	0.1180	0.0801
8	13744.0269	3.560776e+08	18870.0174	0.9465	0.1050	0.0811
9	13230.3240	4.291768e+08	20716.5819	0.9227	0.1148	0.0791
Mean	14566.2730	7.011655e+08	24806.0347	0.8969	0.1222	0.0855
Std	1607.3303	6.650111e+08	9264.2400	0.0777	0.0249	0.0106

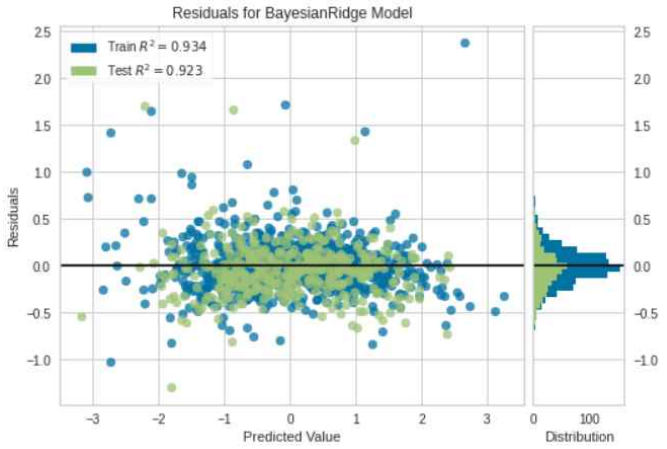
4.8 Blend Tuned Models

```
blend_all = blend_models(estimator_list = [huber, omp, ridge, br])
```

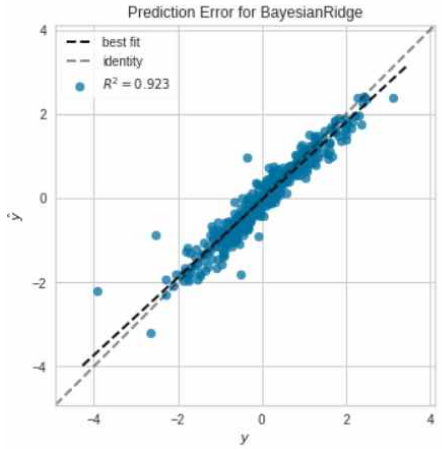
Fold	MAE	MSE	RMSE	R2	RMSLE	MAPE
0	17894.9130	2.701781e+09	51978.6622	0.6633	0.1792	0.1038
1	14795.2990	5.629558e+08	23726.6903	0.9152	0.1128	0.0790
2	12958.2684	3.572251e+08	18900.4007	0.9228	0.1134	0.0819
3	14177.6594	5.151080e+08	22695.9909	0.8978	0.1019	0.0770
4	14437.0594	5.584872e+08	23632.3341	0.9440	0.1454	0.0961
5	15479.4276	7.731938e+08	27806.3623	0.8937	0.1131	0.0856
6	12718.1383	3.262350e+08	18061.9762	0.9235	0.0924	0.0717
7	14673.5174	4.981560e+08	22319.4096	0.9302	0.1144	0.0805
8	13003.9135	3.307605e+08	18186.8227	0.9503	0.1017	0.0769
9	13165.2996	3.903719e+08	19757.8325	0.9297	0.1128	0.0793
Mean	14330.3496	7.014275e+08	24706.6482	0.8970	0.1187	0.0832
Std	1483.0287	6.793821e+08	9539.8646	0.0797	0.0241	0.0092

4.9 Evaluate Bayesian Ridge Model

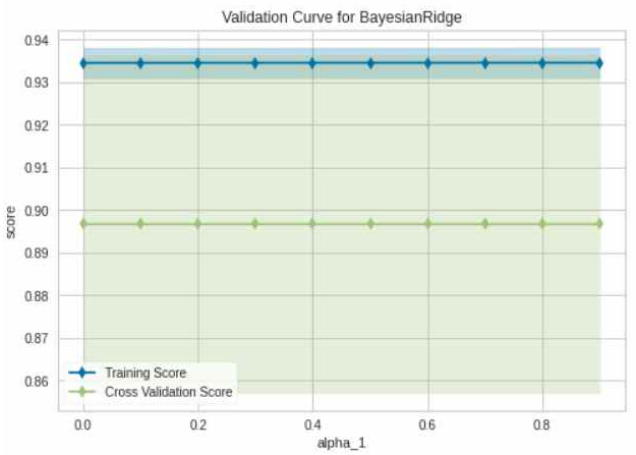
```
plot_model(br, plot = 'residuals')
```



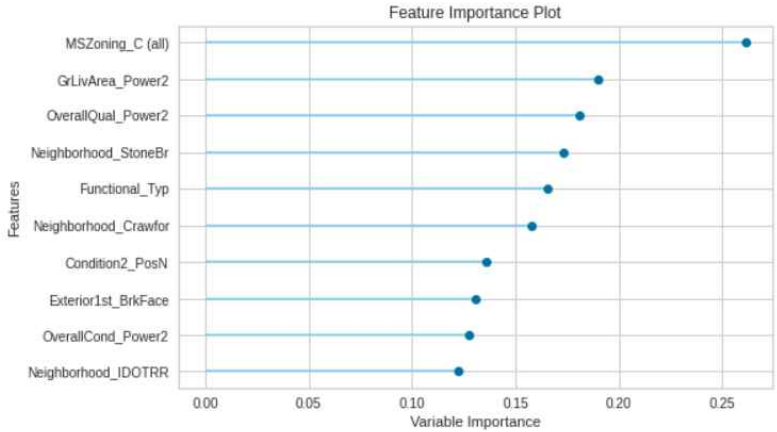

```
plot_model(br, plot = 'error')
```




```
plot_model(br, plot = 'vc')
```

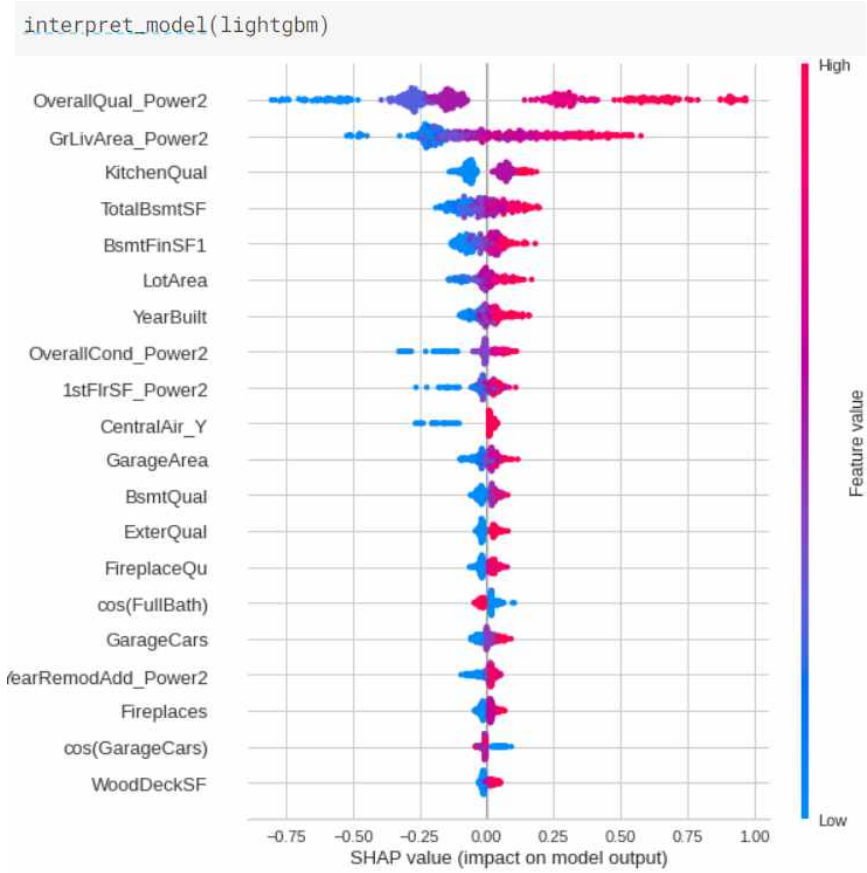



```
plot_model(br, plot = 'feature')
```



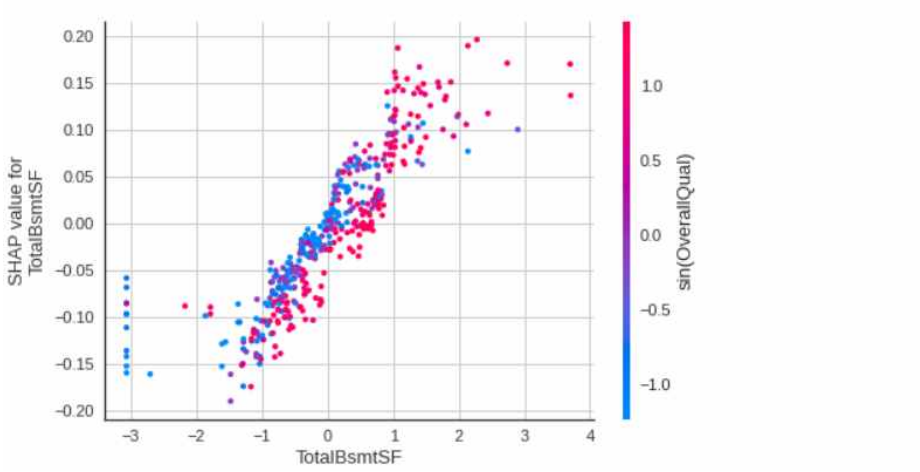
4.10 Interpret LightGBM Model

Interpret model
Interpret_model
: 모델을 해석
  훈련된 모델 객체와 플롯 유형을 문자열로 받음
  해석은 SHAP를 기반으로 구현되며 트리 기반 모델에서만 사용 가능
*Pycaret.classification, pycaret.regression 모듈에서만 사용 가능
*SHAP : SHapley Addictive exPlanatoins

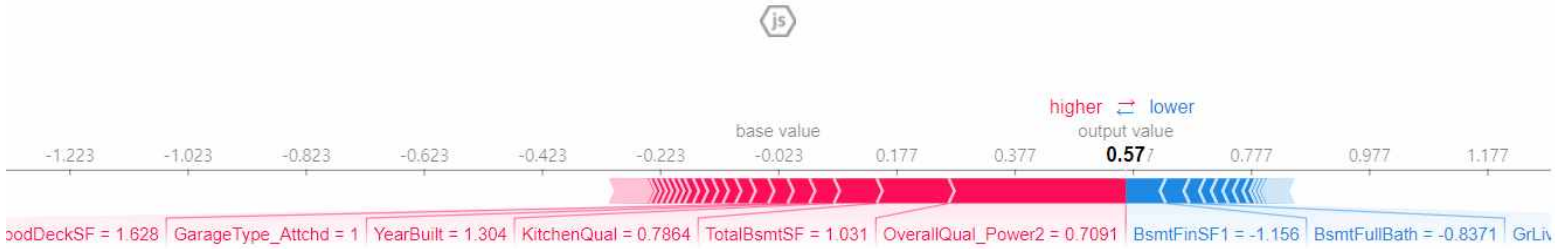


4.10 Interpret LightGBM Model

```
interpret_model(lightgbm, plot = 'correlation', feature = 'TotalBsmtSF')
```




```
interpret_model(lightgbm, plot = 'reason', observation = 0)
```



4.11 Finalize Blender and Predict test dataset

```
predictions = predict_model(final_blender, data = test)
predictions.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal
0	1461	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	MnPrv	NaN	0
1	1462	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	Gar2	12500
2	1463	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	MnPrv	NaN	0
3	1464	60	RL	78.0	9978	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0
4	1465	120	RL	43.0	5005	Pave	NaN	IR1	HLS	AllPub	...	0	NaN	NaN	NaN	0

05. Stacking Regressions

EWHA EURON

5.1 Stacking Regression을 이용한 주택 가격 예측

<목표>: 주거용 주택의 거의 모든 측면을 설명하는 79개의 설명 변수를 사용해
각각의 주택의 가격을 예측하는 것. (Target = SalePrice)

```
train.head(5)
```
✓ 0.1s

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...

5 rows × 81 columns

주의할 점:
train data를 보면 target와 ID를 제외하고도
column이 80개나 되고, NULL 값이 많은 칼럼도 존재한다.
-> 데이터 전처리

EWHA EURON

5.2 데이터 전처리

데이터 전처리

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
%matplotlib inline
import matplotlib.pyplot as plt  # Matlab-style plotting
import seaborn as sns
color = sns.color_palette()
sns.set_style('darkgrid')
import warnings
def ignore_warn(*args, **kwargs):
    pass
warnings.warn = ignore_warn #ignore annoying warning (from sklearn and seaborn)


from scipy import stats
from scipy.stats import norm, skew #for some statistics
```

```
train = pd.read_csv(r'C:\temp\train_hp.csv')
test = pd.read_csv(r'C:\temp\test_hp.csv')
```

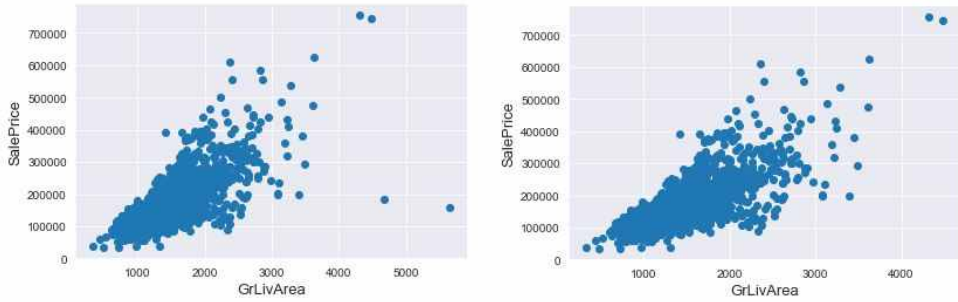
```
train.drop("Id", axis = 1, inplace = True)
test.drop("Id", axis = 1, inplace = True)
```

데이터를 불러오고 필요없는 칼럼인 ID를 drop을 이용해서 지워준다.

이상치 처리

```
fig, ax = plt.subplots()
ax.scatter(x = train['GrLivArea'], y = train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```

```
#Deleting outliers
train = train.drop(train[(train['GrLivArea']>4000) & (train['SalePrice']<300000)].index)

#Check the graphic again
fig, ax = plt.subplots()
ax.scatter(train['GrLivArea'], train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```



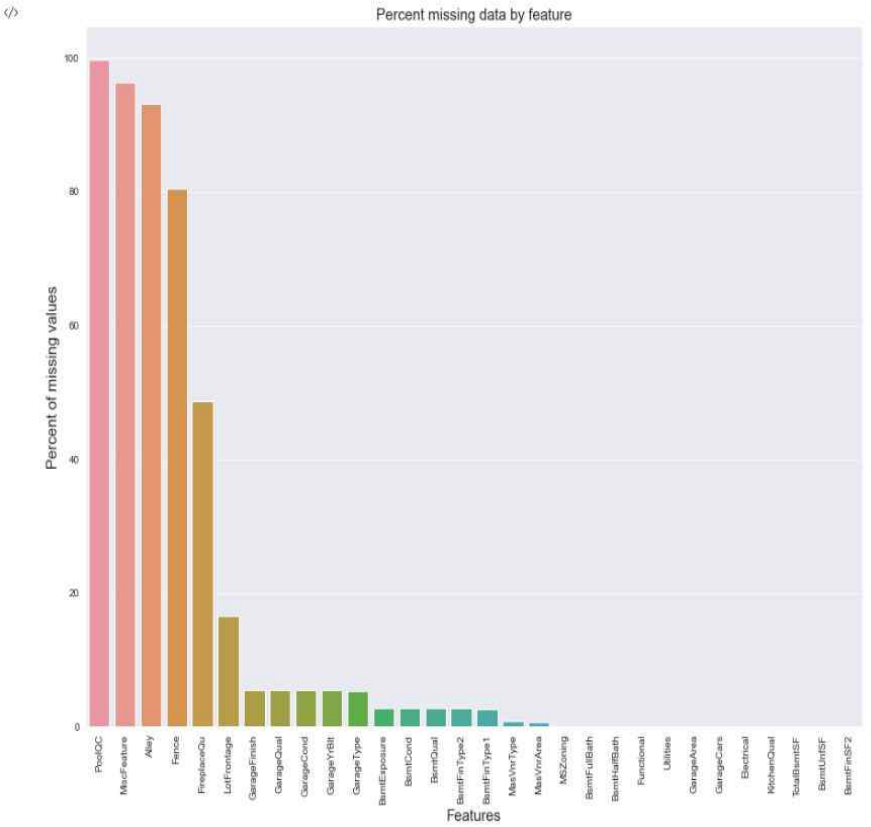
scatter plot을 그려봤을 때 outlier가 있음을 알 수 있다. 하지만 과도한 삭제는 정확성을 떨어트릴 수 있으므로 눈에 보이는 것만 삭제한다

5.2 데이터 전처리

결측치 처리

```
ntrain = train.shape[0]
ntest = test.shape[0]
y_train = train.SalePrice.values
all_data = pd.concat((train, test)).reset_index(drop=True)
all_data.drop(['SalePrice'], axis=1, inplace=True)

all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]

f, ax = plt.subplots(figsize=(15, 12))
plt.xticks(rotation='90')
sns.barplot(x=all_data_na.index, y=all_data_na)
plt.xlabel('Features', fontsize=15)
plt.ylabel('Percent of missing values', fontsize=15)
plt.title('Percent missing data by feature', fontsize=15)
```

결측값의 비율을 먼저 살펴보고 칼럼에 맞춰서
변환을 해준다



5.2 데이터 전처리

결측치 처리

1. 숫자로 이루어져 있지만 범주형이 맞을 때

```
# 연속형 변수를 범주형 변수로 바꾸기

#MSSubClass=The building class
all_data['MSSubClass'] = all_data['MSSubClass'].apply(str)


#Changing OverallCond into a categorical variable
all_data['OverallCond'] = all_data['OverallCond'].astype(str)
```

2. NULL = 없다 의 의미일 때

```
all_data["PoolQC"] = all_data["PoolQC"].fillna("None")
all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
all_data["Alley"] = all_data["Alley"].fillna("None")
all_data["Fence"] = all_data["Fence"].fillna("None")
all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")
```

3. 연속형인 경우 median으로 채워주기

```
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(
    lambda x: x.fillna(x.median()))
```

4. 그 외

```
for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')

for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)

for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'BsmtHalfBath'):
    all_data[col] = all_data[col].fillna(0)

for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')

all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)

all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode()[0])

all_data = all_data.drop(['Utilities'], axis=1)

all_data["Functional"] = all_data["Functional"].fillna("Typ")

all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[0])

all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode()[0])

all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode()[0])
all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode()[0])

all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode()[0])

all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

EWHA EURON

5.2 데이터 전처리

로그변화 / label encoding

```
# target 값 로그변화

train['SalePrice'] = np.log1p(train['SalePrice'])
```

- ✓ Label Encoding의 경우, 캐글 노트북에는 object타입의 칼럼을 직접 입력했는데, 그 외에도 object타입의 칼럼이 남아있어서 select_dtypes("object")를 이용하여서 encoding 해주었음

```
all_data.select_dtypes("object").columns
```

✓ 0.1s

```
Index(['MSSubClass', 'Street', 'Alley', 'LotShape', 'LandContour', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallCond', 'RoofStyle', 'RoofMatl', 'Exterior1st',
       'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
       'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
       'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
       'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
       'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature', 'MoSold',
       'YrSold', 'SaleType', 'SaleCondition'],
      dtype='object')
```

```
from sklearn.preprocessing import LabelEncoder
cols = all_data.select_dtypes("object").columns

# process columns, apply LabelEncoder to categorical features
for c in cols:
    lbl = LabelEncoder()
    lbl.fit(list(all_data[c].values))
    all_data[c] = lbl.transform(list(all_data[c].values))

# shape
print('Shape all_data: {}'.format(all_data.shape))
```

EWHA EURON

5.3 RobustScaler()

라쏘 모델은 이상치에 매우 민감하게 반응한다 -> RobustScaler() 사용

RobustScaler()은 중앙값을 제거하고 Quantile 범위에 따라 데이터를 스케일링하는 기법으로,

StanderScaler보다 동일한 값을 더 넓게 분포시킬 수 있다.

즉, 목표변수 y값을 분류나 예측하는데 있어 산포가 더 크기 때문에 설명변수로서 더 유용할 수 있다고 추정가능

```
from sklearn.preprocessing import RobustScaler

X = [[ 1., -2.,  2.],
     [ -2.,  1.,  3.],
     [ 4.,  1., -2.]]

transformer = RobustScaler().fit(X)
transformer
RobustScaler()
transformer.transform(X)
```

```
array([[ 0. , -2. ,  0. ],
       [-1. ,  0. ,  0.4],
       [ 1. ,  0. , -1.6]])
```

참고: https://wikidocs.net/89704

5.3 RobustScaler()

기본 스케일링 방법 비교

① MinMaxSclaer는 매우 다른 스케일의 범위를 0과 1사이로 변환

② StandardScalar는 각 특성의 평균을 0, 분산을 1로 변경하여 모든 특성이 같은 크기를 가지게 함

③ RobustScaler는 특성들이 같은 스케일을 갖게 되지만 평균대신 중앙값을 사용

==> 극단값에 영향을 받지 않음

④ Nomalizer는 uclidian의 길이가 1이 되도록 데이터 포인트를 조정

==> 각도가 중요할 때 사용

참고: https://wikidocs.net/89704

5.4 KRR (Kernel Ridge Regression)

기존 Ridge Regression: $RSS(\lambda) = (y - X\beta)^T(y - X\beta) + \lambda\beta^T\beta$

Kernel Ridge Regression: $RSS(\lambda) = (y - \phi\beta)^T(y - \phi\beta) + \lambda\beta^T\beta$

- 기존의 Ridge Regression은 아래의 같은 RSS(Residual sum of squares)를 최소화 하는 것이고, Kernel Ridge Regression은 X를 Mapping한 ø에 대해 위의 식을 최소화하는 Parameter를 구하는 것이고 위와 같은 RSS를 최소화하도록 한다

5.5 모델 학습, 예측, 평가

Stacking models:
Averaging base models 평균 베이스 모델들의 합을 통한 모델 구상

LASSO

```
# LASSO Regression

lasso = make_pipeline(RobustScaler(), Lasso(alpha =0.0005, random_state=1))
```

Elastic Net Regression

```
#Elastic Net Regression

ENet = make_pipeline(RobustScaler(),
                     ElasticNet(alpha=0.0005, l1_ratio=.9, random_state=3))
```

Kernel Ridge Regression

```
# Kernel Ridge Regression

KRR = KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5)
```

Gradient Boosting Regression

```
#Gradient Boosting Regression

GBoost = GradientBoostingRegressor(n_estimators=3000, learning_rate=0.05,
                                   max_depth=4, max_features='sqrt',
                                   min_samples_leaf=15, min_samples_split=10,
                                   loss='huber', random_state =5)
```

XGBoost

```
#XGBOOST

model_xgb = xgb.XGBRegressor(colsample_bytree=0.4603, gamma=0.0468,
                             learning_rate=0.05, max_depth=3,
                             min_child_weight=1.7817, n_estimators=2200,
                             reg_alpha=0.4640, reg_lambda=0.8571,
                             subsample=0.5213, silent=1,
                             random_state =7, nthread = -1)
```

LightGBM

```
#LGBM

model_lgb = lgb.LGBMRegressor(objective='regression',num_leaves=5,
                              learning_rate=0.05, n_estimators=720,
                              max_bin = 55, bagging_fraction = 0.8,
                              bagging_freq = 5, feature_fraction = 0.2319,
                              feature_fraction_seed=9, bagging_seed=9,
                              min_data_in_leaf =6, min_sum_hessian_in_leaf = 11)
```

EWHA EURON

5.5 모델 학습, 예측, 평가

Base 모델들과 모델 평가

```
#Validation function

n_folds = 5

def rmsle_cv(model):
    kf = KFold(n_folds, shuffle=True, random_state=42).get_n_splits(train.values)
    rmse= np.sqrt(-cross_val_score(model, train.values, y_train, scoring="neg_mean_squared_error", cv = kf))
    return(rmse)
```

```
score = rmsle_cv(lasso)
print("Lasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))

score = rmsle_cv(ENet)
print("ElasticNet score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))

score = rmsle_cv(KRR)
print("Kernel Ridge score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))

score = rmsle_cv(GBoost)
print("Gradient Boosting score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))

score = rmsle_cv(model_xgb)
print("Xgboost score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))

score = rmsle_cv(model_lgb)
print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))
```

사이킷런의 cross_val_score에는 Shuffle 기능이 없으므로 KFold 함수를 사용해서 데이터 shuffle 까지 수행한다.

MODEL	Score.mean	Score.std
LASSO	0.1237	0.0048
Elastic Net Regression	0.1236	0.0048
Kernel Ridge Regression	1.4031	0.8634
Gradient Boosting Regression	0.1208	0.0087
XGBoost	0.1192	0.0046
LightGBM	0.1196	0.0061

5.5 모델 학습, 예측, 평가

Base models의 예측값 평균으로 계산

기본 모델을 평균화 하는 간단한 방법으로 모델을 Stacking.

여러 모델을 동시에 사용하고 해당 모델들의 예측값을 평균내어서 최종 예측값으로 사용한다.

```
class AveragingModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, models):
        self.models = models

    # we define clones of the original models to fit the data in
    def fit(self, X, y):
        self.models_ = [clone(x) for x in self.models]

        # Train cloned base models
        for model in self.models_:
            model.fit(X, y)

        return self

    #Now we do the predictions for cloned models and average them
    def predict(self, X):
        predictions = np.column_stack([
            model.predict(X) for model in self.models_
        ])
        return np.mean(predictions, axis=1)
```

모델: (ENet, GBoost, KRR, lasso)

```
averaged_models = AveragingModels(models = (ENet, GBoost, KRR, lasso))

score = rmsle_cv(averaged_models)
print(" Averaged base models score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
score: 0.3656 (0.1980)
```

모델: (model_xgb, model_lgb, lasso, GBoost)

```
averaged_models = AveragingModels(models = (model_xgb, model_lgb, lasso, GBoost))

score = rmsle_cv(averaged_models)
print(" Averaged base models score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
score: 0.1147 (0.0058)
```

EWHA EURON

5.5 모델 학습, 예측, 평가

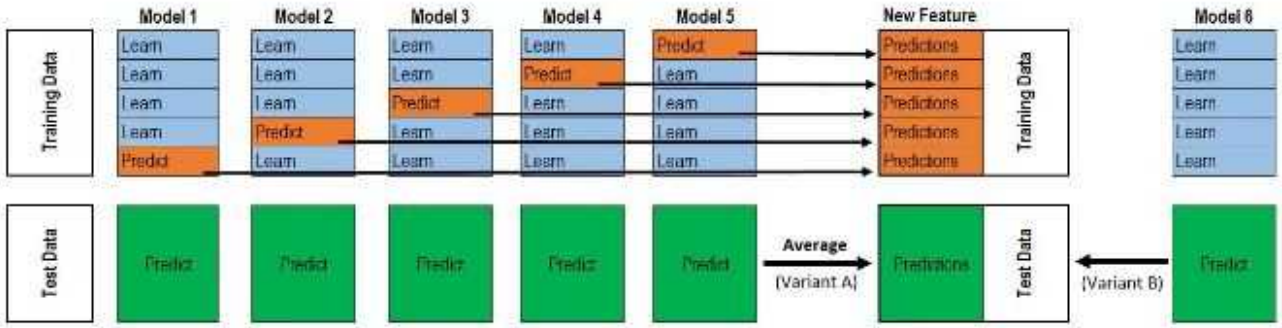
Adding a Meta-model

① training set를 train과 holdout으로 나눈다

② train 데이터로 base 모델들을 여러 번 train한다

③ holdout 데이터로 base 모델들을 여러 번 train한다 (out-of-folds 예측)

④ out-of-folds 예측을 input으로, target variable을 output을 사용한 예측으로 meta-model을 train한다



참고: https://www.kaggle.com/getting-started/18153#post103381

5.5 모델 학습, 예측, 평가

Adding a Meta-model

```
class StackingAveragedModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, base_models, meta_model, n_folds=5):
        self.base_models = base_models
        self.meta_model = meta_model
        self.n_folds = n_folds

    # 다시 원본 모델의 복제본에 데이터 형식을 맞춤
    def fit(self, X, y):
        self.base_models_ = [list() for x in self.base_models]
        self.meta_model_ = clone(self.meta_model)
        kfold = KFold(n_splits=self.n_folds, shuffle=True, random_state=156)

        # 복제된 base 모델들을 학습시킨 뒤, out-of-fold 예측
        # meta-model을 훈련시키기 위한 과정
        out_of_fold_predictions = np.zeros((X.shape[0], len(self.base_models)))
        for i, model in enumerate(self.base_models):
            for train_index, holdout_index in kfold.split(X, y):
                instance = clone(model)
                self.base_models_[i].append(instance)
                instance.fit(X[train_index], y[train_index])
                y_pred = instance.predict(X[holdout_index])
                out_of_fold_predictions[holdout_index, i] = y_pred

        # out-of-fold 예측을 새로운 features로 추가하여 새로운 meta-model학습
        self.meta_model_.fit(out_of_fold_predictions, y)
        return self

    # 테스트 데이터에 대한 모든 기본 모델의 예측을 수행하고,
    # 평균 예측을 메타 모델에 의해 수행되는 최종 예측을 위한 메타 기능으로 사용
    def predict(self, X):
        meta_features = np.column_stack([
            np.column_stack([model.predict(X) for model in base_models]).mean(axis=1)
            for base_models in self.base_models_ ])
        return self.meta_model_.predict(meta_features)
```

모델: (ENet, GBoost, KRR, lasso)

```
stacked_averaged_models = StackingAveragedModels(base_models = (ENet, GBoost, KRR),
                                                 meta_model = lasso)

score = rmsle_cv(stacked_averaged_models)
print("Stacking Averaged models score: {:.4f} ({:.4f})".format(score.mean(), score.std()))
```

```
score: 0.1145 (0.0059)
```

모델: (model_xgb, model_lgb, lasso, GBoost)

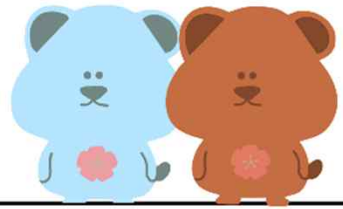
```
stacked_averaged_models = StackingAveragedModels(base_models = (model_xgb, model_lgb, lasso, GBoost),
                                                 meta_model = lasso)

score = rmsle_cv(stacked_averaged_models)
print("Stacking Averaged models score: {:.4f} ({:.4f})".format(score.mean(), score.std()))
```

```
score: 0.1140 (0.0055)
```

✓ Average 모델과 비교했을 때,
메타 모델을 추가함으로써
성능이 좋아진 것을 알 수 있다

EWHA EURON

THANK YOU



EWHA EURON