5. 회귀 (2)

goblurry · 방금 전

데이터분석 머신러닝 회귀분석

📊 파이썬 머신러닝 완벽 가이드 (위키북스, 개정 2판) 교재의 **5.9, 5.10장**을 바탕으로 학습한 내용을 정리한 포스트입니다. https://github.com/goblurry/ML-Study 에서 예시 코드를 확인할 수 있습니다.

이전 포스팅에서 학습한 개념을 바탕으로 실습을 진행합니다.

1. 자전거 대여 수요 예측 (https://www.kaggle.com/c/bike-sharing-demand/data)
2. 캐글 주택 가격: 고급 회귀 기법 (https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques/data)

[기본적인 회귀 개념]

- 회귀: 여러 개의 독립변수와 한 개의 종속변수 간의 상관관계를 모델링하는 기법
- 머신러닝 회귀 예측: 주어진 피처와 결정 값 데이터 기반에서 학습을 통해 최적의 회귀 계수를 찾는 것
  (회귀 계수: 독립변수 값에 영향 미치는 값들)
  - 머신러닝 관점에서 독립변수는 피처, 종속변수는 결정 값
- 회귀 분석: 데이터 값이 평균과 같은 일정한 값으로 돌아가려는 경향을 이용한 통계학 기법

5.9 자전거 대여 수요 예측

이 데이터셋은 날짜/시간, 기온, 습도, 풍속 등 정보를 기반으로 1시간 간격 동안의 자전거 대여 횟수가 기재되어 있다. 이는 count라는 칼럼명으로 정의되어 있으며, 결정 값(종속 변수)이다.

먼저 데이터셋을 데이터프레임으로 만들어 bike_df에 저장한다.

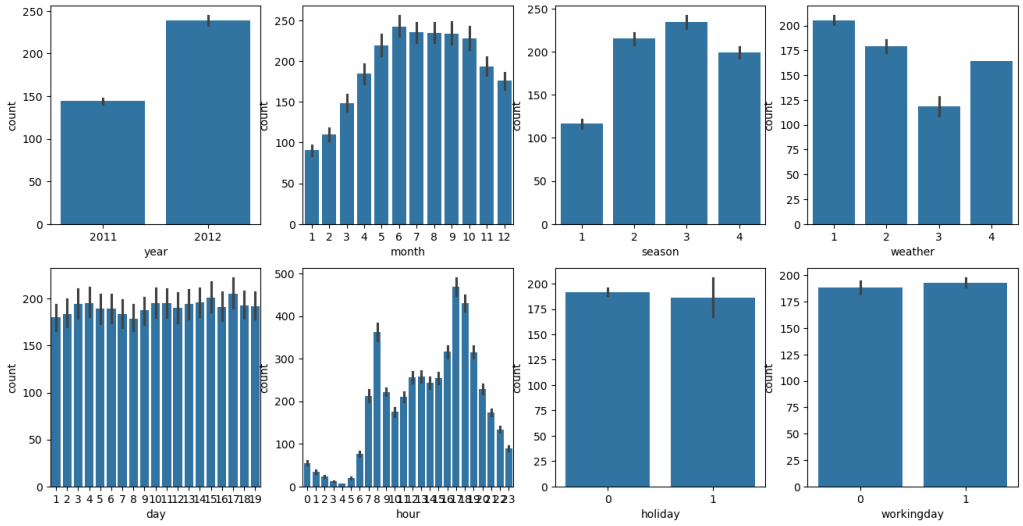
	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	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_df.info() 실행 시, datetime을 제외한 모든 칼럼의 데이터 타입은 숫자형이나, datetime은 object 타입이다. 이 칼럼은 **년-월-일 시:분:초** 형식으로 되어 있어 가공이 필요하다.
판다스에서 이 칼럼을 4개의 속성으로 분리하기 위해, 문자열을 **'datetime' 타입**으로 변경한다. 이때 데이터셋에서 날짜를 의미하는 datetime 칼럼과 다르다. 이름이 같을 뿐...
메서드: `apply(pd.to_datetime)`

```
#문자열 > datetime 타입
# 기존의 'datetime' 컬럼을 datetime 타입으로 변환해서 다시 같은 자리에 덮어쓰기
bike_df['datetime'] = bike_df.datetime.apply(pd.to_datetime)

# datetime 타입에서 년, 월, 일, 시간 추출
# x라는 datetime 객체에서 (년, 월, 일, 시간)만 꺼내라는 의미의 람다
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)
```

그럼 이제 year, month, day, hour 칼럼이 추가된다. datetime 칼럼은 불필요해지므로 삭제한다. (drop 메서드 활용)

다음으로, 칼럼별 값에 따른 count 합을 seaborn의 barplot을 사용해 시각화한다. 이는 주요 칼럼별 target 값인 count가 어떻게 분포되어 있는지 확인하기 위함이다. (year, month, season, weather, day, hour, holiday, workingday)



위와 같이 개별 칼럼에 따른 count 값을 비교할 수 있다. 2011년보다 2012년이 많고, 6-9월의 대여 수가 1-3월의 대여 수보다 많고... 이런 식으로 **각 범주형 변수(연도, 월, 계절, 날씨 등)**에 따라 자전거 대여 횟수(count)가 평균적으로 어떻게 달라지는지를 비교** 한다.

다양한 회귀 모델 데이터셋에 적용해 예측 성능 측정하기

캐글에서 요구한 회귀 모델 성능 평가 지표: RMSLE (Root Mean Square Log Error)

사이킷런이 이 RMSLE를 지원하지 않아 직접 함수를 만든다.

- RMSLE: 오류값 로그에 대한 RMSE
- RMSE: MSE에 루트 씌운 값
- MSE: Mean Squared Error. (실제값-예측값)^2 후 평균

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

# log 값 변환 시 NaN 등의 이슈로 log()가 아닌 log1p() 이용해 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() 사용해 RMSE 계산
def rmse(y, pred):
    return np.sqrt(mean_squared_error(y, pred))

# MAE, RMSE, RMSLE 모두 계산
def evaluate_regr(y, pred):
    rmsle_val = rmsle(y, pred)
    rmse_val = rmse(y, pred)
    # MAE: 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_va
```

로그 변환, 피처 인코딩과 모델 학습/예측/평가

회귀 모델을 이용해 자전거 대여 횟수를 예측하기 앞서 데이터셋 처리가 필요하다.
1. 결괏값이 **정규분포**로 되어 있는지?
2. **카테고리형 회귀 모델의 경우 원핫 인코딩**으로 인코딩하기.

로그 변환 먼저, 선형회귀 객체를 이용해 회귀를 예측해 본다. 사진에 보이는 것과 같이 RMSe가 140.9가 출력되는데, 이는 평균적으로 한 시간마다 예측이 약 ±141대 정도 틀리고 있다는 뜻으로, 비교적 큰 오류값이다. 이렇게 회귀에서 예측 오류가 클 때에는 target 값이 왜곡된 형태인 건 아닌지 가장 먼저 확인해 보아야 한다. pandas의 hist() 메서드를 활용해 체크해 본다. 그럼 이런 형태의 그래프가 출력되는데, count 칼럼값이 0에서 200 사의 값에 왜곡되어 있음을 알 수 있다. 이럴 땐 **로그를 적용해 정규분포 형태**로 변환한다. 넘파이의 **log1p**를 이용해 새롭게 학습하고 예측한 뒤에, 다시 expm1() 함수 -로그함수의 역함수-를 적용해 원래 스케일로 복구하면 된다. ``` y_log_transform = np.log1p(y_target) y_log_transform.hist() ``` 그럼 이제 로그를 적용해서 0-1000 범위에 있던 값들이 0-7 범위로 압축되고, 0-200에 몰리던 데이터가 2-5 사이에 퍼지게 된다. 이렇게 로그 적용 시 값의 스케일을 압축해 큰 값들의 영향력을 완화시킨다.

다음으로 위의 np.log1p(y_target)를 활용해 재학습한 후 평가를 수행한다.

```
# 이를 사용해 다시 학습한 후 평가 수행
y_target_log = np.log1p(y_target)

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 데이터셋의 타겟 값을 로그 변환했으므로 다시 expm1(log1p의 역함수, exponential minus 1) 사용해 원래 스케일로 변환
y_test_exp = np.expm1(y_test)

# 예측값도 다시 변환
pred_exp = np.expm1(pred)

evaluate_regr(y_test_exp, pred_exp)
```

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

[최초] RMSLE: 1.165, RMSE: 140.900, MAE: 105.924
[로그 변환 후] RMSLE: 1.017, RMSE: 162.594, MAE: 109.286
RMSLE는 줄었지만 RMSE는 늘었음 > **개별 피처들의 인코딩 적용하기**

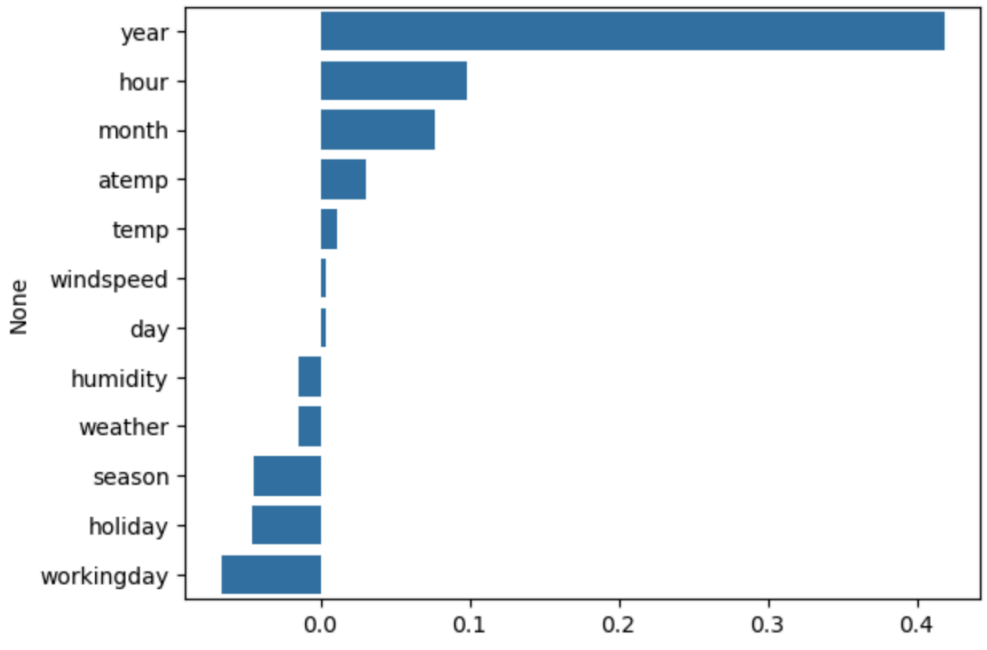
인코딩

위에서 로그 변환 후 RMSLE는 줄었지만 RMSE가 늘어난 걸 볼 수 있다. 이번에는 개별 피처들에 인코딩을 적용한다.
인코딩은 `get_dummies()` 메서드를 활용한다.

먼저 어떤 피처를 인코딩할지 확인해 보기 위해, 각 피처의 회귀 계숫값을 시각화해보았다.

```
# 각 피처의 회귀 계숫값 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)
```

```
[31]:

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



피처 year, hour, month, season, holiday, workingday에서 회귀 계수 영향도가 상대적으로 높게 출력되었다. year는 2011, 2012, month는 1, 2, ..., 12처럼 숫자값 형태로 의미를 담고 있다. 하지만 이 피처들의 경우 개별 숫자값의 **크기**가 의미를 가지는 것은 아닌, 즉, 카테고리형 피처이다. 사이킷런은 이런 카테고리만을 위한 데이터 타입이 없기 때문에 이를 모두 숫자로 변환해야 한다. 그러나 이렇게 **숫자형 카테고리값**을 선형회귀에 사용하는 경우, 회귀 계수 연산 시에 이 값에 크게 영향을 받는 경우가 발생할 수 있어서 **원-핫 인코딩**을 적용해야 한다. 이때 활용하는 메서드가 앞서 말한 get_dummies이다.

```
# One-hot Encoding
X_features_ohe = pd.get_dummies(X_features, columns=['year', 'month', 'day', 'hour', 'holi
```

이렇게 원핫 인코딩이 적용된 피처 데이터셋을 기반으로 학습을 수행하고 모델별(선형회귀, ridge, lasso)로 평가를 수행한다.

```
# ohe 적용된 피처 데이터셋 기반 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 데이터셋 입력하면 성능 평가 수치 반환하는 함수 정의
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)

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

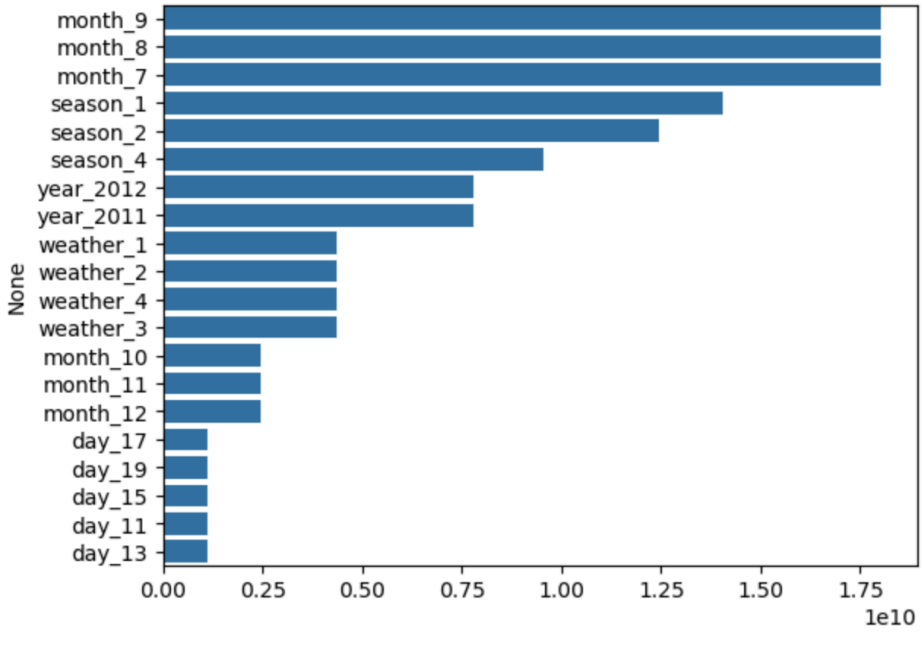
여기서 이제 회귀 계수가 높은 피처를 다시 시각화한다. 인코딩 이후 피처가 늘어났으므로 회귀 계수 상위 20개 피처를 다시 추출한다. (왜 늘어나냐: 하나의 범주형 변수-column-를 그 안의 고유한 값 개수만큼의 새 피처로 나누기 때문이다. 즉, season이라는 범주 안에서 봄=1, 여름=2, .. 이런 식으로 나눴다면 원핫 인코딩 이후 season_1, season_2... 이런 식의 4개의 칼럼으로 쪼갠다.)

[36]:

```
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)
```

[36]:

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



여전히 year 관련 피처의 회귀계수 값이 크지만, season과 weather 관련 속성들의 값도 상대적으로 커졌다. 즉, 원핫 인코딩을 통해 피처들의 영향도가 달라지고 모델 성능도 향상되었다. (항상 그런 건 아니지만, 선형 회귀에서 중요한 **카테고리성 피처들을 원핫 인코딩으로 변환**하는 건 성능에 중요한 영향을 미칠 수 있다.)

이렇게 타겟의 로그 변환값+원핫 인코딩된 피처 데이터셋을 이용해 랜덤 포레스트, GBM, XGBoost, LightGBM에 적용해 성능을 평가한 결과,

```
### RandomForestRegressor ###
RMSLE: 0.355, RMSE: 50.358, MAE: 31.139
### GradientBoostingRegressor ###
RMSLE: 0.330, RMSE: 53.343, MAE: 32.743
### XGBRegressor ###
RMSLE: 0.342, RMSE: 51.732, MAE: 31.251
### LGBMRegressor ###
RMSLE: 0.319, RMSE: 47.215, MAE: 29.029
```

가 출력되었다. 이전의 선형회귀 모델과 비교했을 때 성능이 개선됐음을 알 수 있다.

5.10 캐글 주택 가격: 고급 회귀 기법

RMSLE를 기반으로 미국 아이오와 주의 Ames 지방의 주택 가격 정보 저장하는 데이터 세트. 비싼 주택일수록 예측 결과 오류가 전체 오류에 미치는 비중이 높으므로 이를 상쇄하기 위해 오류값을 로그 변환한 RMSLE를 성능 평가 지표로 이용한다.

마찬가지로 데이터 사전처리 (정규분포인지 확인하고 로그 변환, 원핫 인코딩) 후 선형회귀 모델 학습/예측/평가, 회귀 트리 모델 학습/예측/평가, 회귀 모델들의 예측 결과 혼합을 통한 최종 예측 과정으로 진행한다.

```
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	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	MoSc
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	

3 rows × 81 columns

이 데이터셋의 타겟 값은 SalePrice 값이다. 이 예제는 데이터 가공을 많이 수행하기 때문에 원본 데이터프레임은 보관하고 복사해서 데이터를 가공한다.

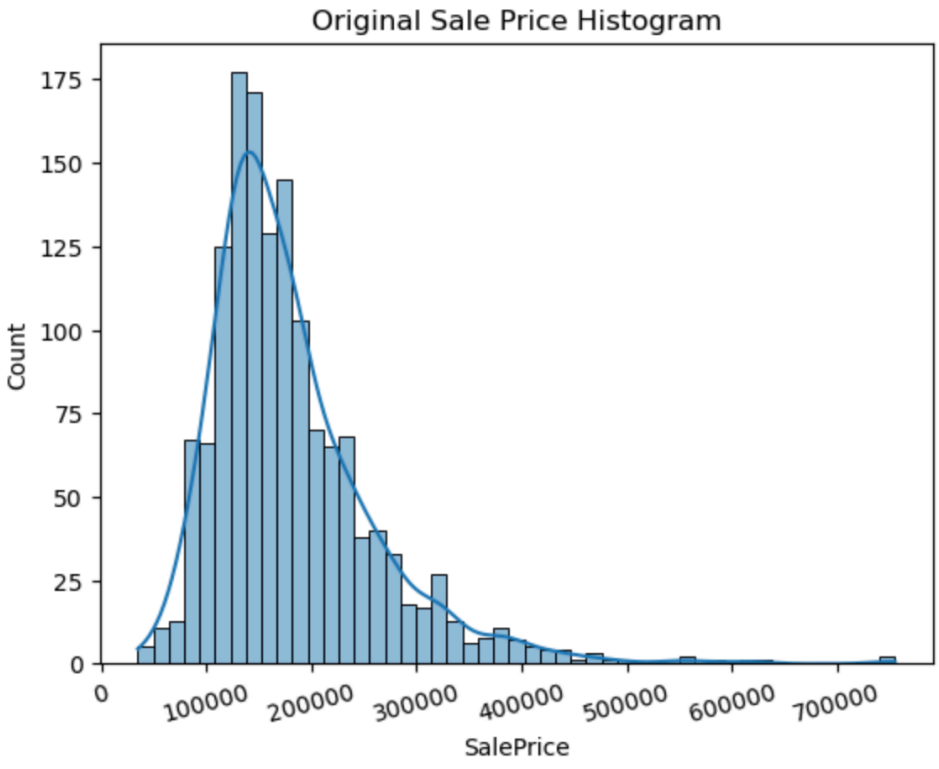
```
# target: SalePrice
print('데이터 세트의 Shape:', house_df.shape)
print('\n전체 feature 들의 type \n',house_df.dtypes.value_counts())
isnull_series = house_df.isnull().sum()
print('\nNull 컬럼과 그 건수:\n ', isnull_series[isnull_series > 0].sort_values(ascending=False))
```

```
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
GarageType         81
GarageYrBlt        81
GarageFinish       81
GarageQual         81
GarageCond         81
BsmtExposure       38
BsmtFinType2       38
BsmtFinType1       37
BsmtCond           37
BsmtQual           37
MasVnrArea          8
MasVnrType          8
Electrical          1
dtype: int64
```

이 데이터셋은 문자형 데이터도 많고, 데이터 양에 비해 null 값이 많은 피처도 있어서, 전체 데이터 중 null 값이 너무 많은 피처는 드롭한다.

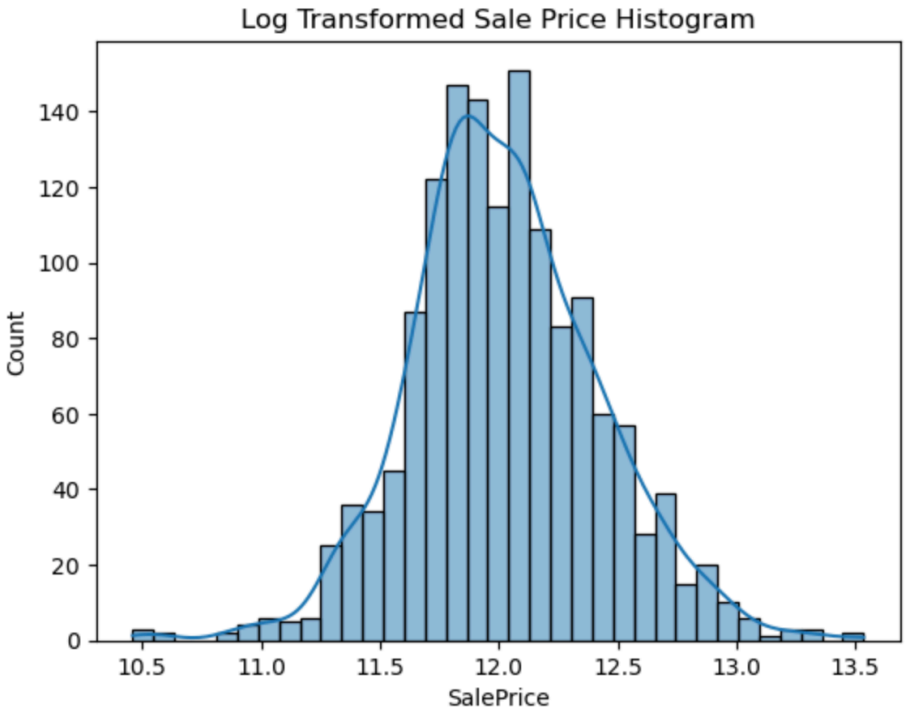
먼저 회귀 모델을 적용하기 전에 **타깃 값의 분포도가 정규 분포인지 확인: `hist()`**

```
# 타깃 값 정규 분포인지 확인: hist()
plt.title('Original Sale Price Histogram')
plt.xticks(rotation=15)
sns.histplot(house_df['SalePrice'], kde=True)
plt.show()
```



데이터 값의 분포가 왼쪽으로 치우친, 정규분포에서 벗어난 형태이다. 넘파이의 log1p를 이용해 로그 변환을 적용한다.

```
# 로그 변환해 정규 분포 형태로 변환하기
plt.title('Log Transformed Sale Price Histogram')
log_SalePrice = np.log1p(house_df['SalePrice'])
sns.histplot(log_SalePrice, kde=True)
plt.show()
```



로그 변환을 적용해 SalePrice가 정규분포 형태로 분포함을 확인할 수 있다!

다음으로는 null이 너무 많은 컬럼과 불필요한 컬럼을 삭제하고, 드랍하지 않는 숫자형 null 컬럼은 평균값으로 대체한다.

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

# Null이 너무 많은 컬럼들과 불필요한 컬럼 삭제
house_df.drop(['Id','PoolQC' , 'MiscFeature', 'Alley', 'Fence','FireplaceQu'], axis=1 , 

# 드랍 하지 않는 숫자형 Null 컬럼들은 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
```

이제 문자형 피처를 제외하고는 null 값 없고, 문자형은 전부 **원핫 인코딩**으로 변환한다. get_dummies():

자동으로 문자열 피처를 원핫 인코딩 변환하면서 null 값은 모든 인코딩 값이 0으로 변환되는 방식으로 대체해 주므로 별도의 널값을 대체하는 로직이 불필요하다.

선형 회귀 모델 학습/예측/평가

앞에서 타깃 값인 SalePrice가 로그 변환되었고 예측값도 이 값을 기반으로 예측하므로 원본 SalePrice 예측값의 로그 변환 값이다. 따라서 예측 결과 오류에 RMSE만 적용하면 자동으로 RMSLE가 측정된다.
여러 모델의 로그 변환된 RMSE를 측정하기 위한 함수를 생성한다:

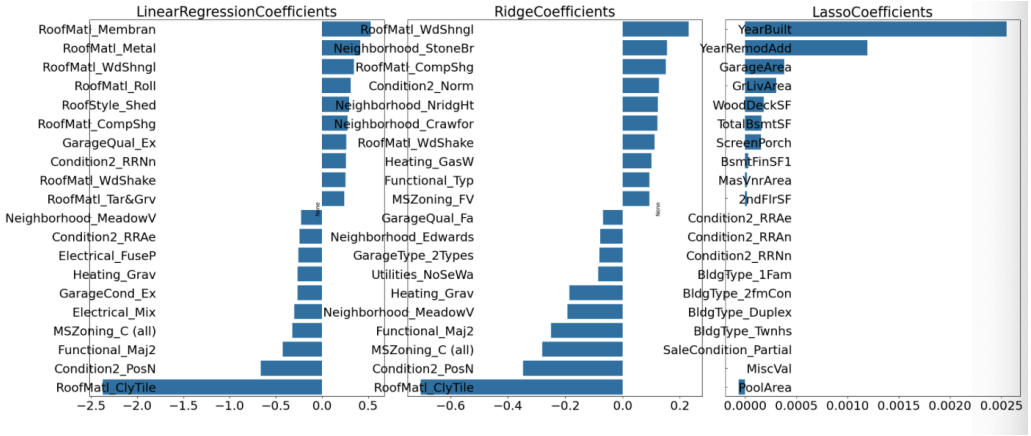
```
# 선형회귀 모델 학습, 예측, 평가

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, r

# 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

Lasso 회귀는 다른 방식보다 성능이 떨어진다. 최적 하이퍼 파라미터 튜닝이 필요하다. 먼저 피처별 회귀 계수를 시각화해서 모델마다 어떤 피처의 회귀 계수로 구성되는지 확인한다. (상하위 10개 피처명과 그 회귀 계수 값을 가지는 판다스 시리즈 객체를 반환)

```
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


def visualize_coefficient(models):
    # 3개 회귀 모델의 시각화를 위해 3개 칼럼 갖는 subplot 생성
    fig, axs = plt.subplots(figsize=(24,10), nrows=1, ncols=3)
    fig.tight_layout()

    # 입력 인자로 받은 list 객체인 models에서 차례로 model 추출해 회귀계수 시각화
    for i_num, model in enumerate(models):
        # 상하위 10개씩 회귀 계수 구하고 판다스 concat으로 결합
        coef_high, coef_low = get_top_bottom_coef(model)
        coef_concat = pd.concat([coef_high, coef_low])
        # ax subplot에 barchar로 표현
        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 회귀계수 시각화
models = [lr_reg, ridge_reg, lasso_reg]
visualize_coefficient(models)
```



상단의 그래프: 모델별 회귀 계수

선형, 릿지의 경우 회귀 계수가 유사한 형태로 분포되어 있으나 **lasso**는 전체적으로 회귀계수 값이 작고 극단적이다.

전체 데이터 세트를 학습/테스트로 분할하지 않고, 5개의 교차 검증 폴드로 분할해 평균 RMSE 측정한다: by `cross_val_score`

```
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: [0.135 0.165 0.168 0.111 0.198]
LinearRegression CV 평균 RMSE 값: 0.155

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
```

여전히 라쏘 모델은 성능이 떨어진다: 릿지와 라쏘 모델에 대해 alpha 하이퍼 파라미터 값 변화시키며 최적값 도출

```
# print_best_params(model, params) : 모델과 하이퍼 파라미터 딕셔너리 객체 받아 최적화 작업 결과 표시하는 함

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_))
```

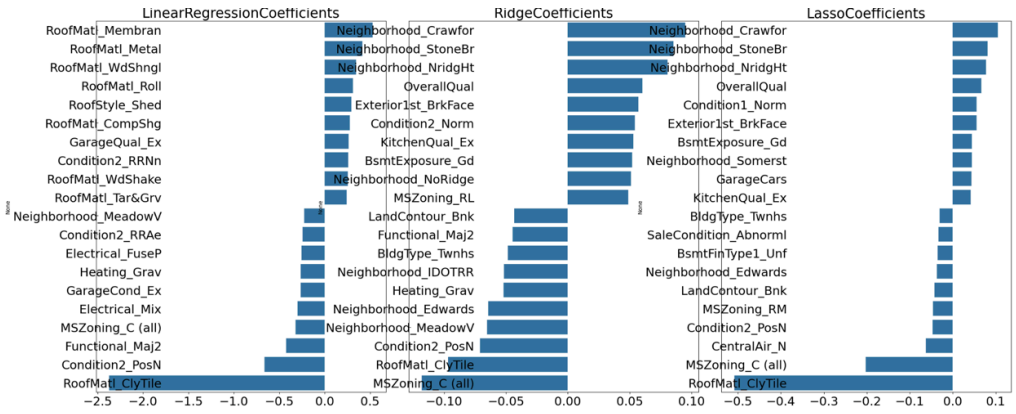
```
    return grid_model.best_estimator_

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}

라쏘: 알파가 0.001에서 최적 평균 RMSE가 0.142 로, 이전의 0.198에 비해 성능이 많이 좋아졌다.
선형 모델에 최적 알파 값을 설정한 뒤 train_test_split()으로 분할된 데이터를 이용해 모델 학습/예측/평가 수행, 회귀 계수 시각화.

```
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)
```

이렇게 위에서 출력된 알파값을 적용해서 다시 예측 및 평가를 수행하면,
LinearRegression 로그 변환된 RMSE: 0.132
Ridge 로그 변환된 RMSE: 0.124
Lasso 로그 변환된 RMSE: 0.12



예측 성능 좋아지고 모델별 회귀 계수도 많이 달라진 걸 확인할 수 있다. 또, 릿지와 라쏘 모델에서 비슷한 피처의 회귀 계수가 높다. 다만 라쏘 모델은 동일한 피처여도 회귀계수 값이 작다... > **데이터셋 가공하여 모델 튜닝 진행하기**

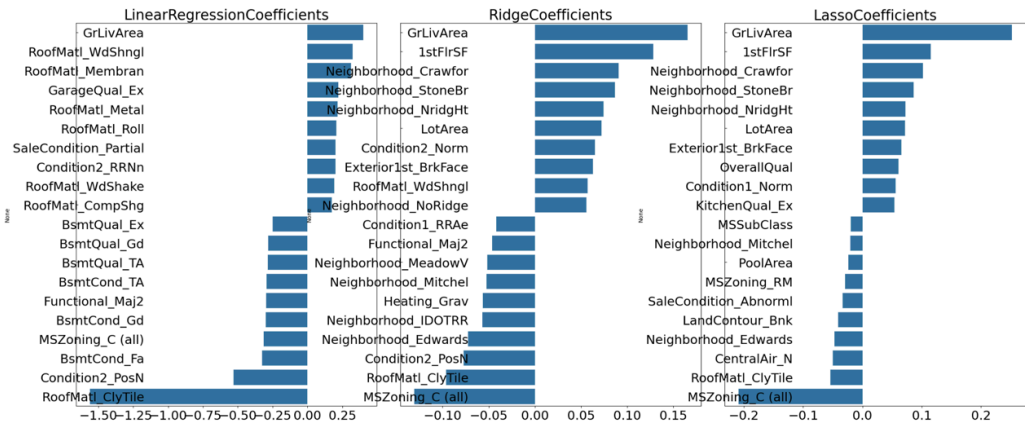
```
# 1. 피처 데이터셋 데이터 분포도
# 숫자형 피처의 데이터 분포도 확인해 분포도가 어느 정도로 왜곡되었는지 확인: 사이파이 stats 모듈의 skew() 메서드
# 일반적으로 1 이상인 경우 왜곡 정도 높다고 판단함. 여기선 로그 변환을 적용해 1 이상 값을 반환하는 피처만 추출.

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))

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

```
# 왜곡 정도 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, r

# 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}

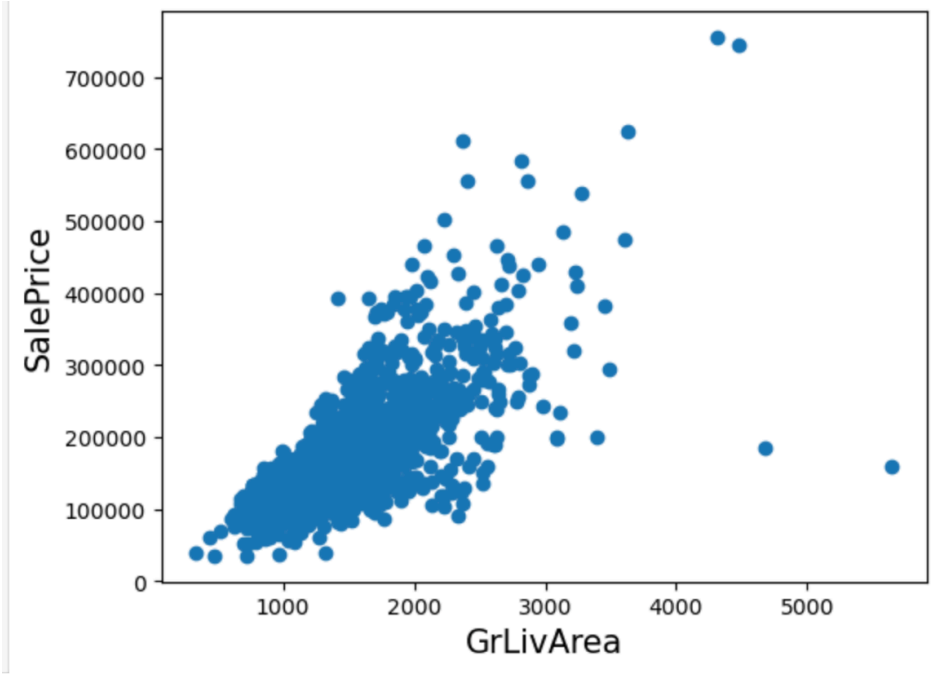
```
# 앞의 최적화 alpha값으로 학습데이터로 학습, 테스트 데이터로 예측 및 평가 수행.
lr_reg = LinearRegression()
lr_reg.fit(X_train, y_train)
ridge_reg = Ridge(alpha=10)
ridge_reg.fit(X_train, y_train)
lasso_reg = Lasso(alpha=0.001)
lasso_reg.fit(X_train, y_train)

# 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



다음으로 이상치 데이터를 분석한다. 회귀 계수 높은 피처의 이상치 데이터 처리가 중요하다: GrLivArea 피처가 세 모델 모두에서 가장 크기 때문에 원본 데이터셋에서 이 GrLivArea와 타겟의 관계를 시각화한다.

```
# 2. 이상치 데이터 분석
# 회귀 계수 높은 피처의 이상치 데이터 처리가 중요하다: GrLivArea 피처가 세 모델 모두에서 가장 큼
# 원본 데이터셋에서 GrLivArea와 SalePrice 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()
```



우측 하단에 이상치 두 개가 보인다. 두 데이터의 GrLivArea가 가장 큰데도 가격은 매우 높다. 이것들을 삭제하자... 이제 데이터 변환이 모두 완료된 house_df_ohe에서 대상 데이터를 필터링한다!!

```
# 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('이상치 레코드 index :', outlier_index.values)
print('이상치 삭제 전 house_df_ohe shape:', house_df_ohe.shape)

# DataFrame의 index를 이용하여 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 index : [523 1298]
이상치 삭제 전 house_df_ohe shape: (1460, 271)
이상치 삭제 후 house_df_ohe shape: (1458, 271)

2개가 삭제된 걸 확인할 수 있다. 갱신된 **house_df_ohe**로 다시 데이터셋을 생성하고 모델 최적화를 수행한다.

```
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, r

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}

가 출력된다. 이 최적화 alpha값으로 학습 데이터로 학습, 테스트 데이터로 예측 및 평가 수행한다.

```
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

회귀 트리 모델 학습/예측/평가

이제 회귀 트리를 이용해 회귀 모델을 생성한다: XGBoost, LightGBM

```
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: 0.1197, 최적 alpha:{'n_estimators': 1000}
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.5, enable_categorical=False,
             gamma=0, gpu_id=-1, importance_type=None,
             interaction_constraints='', learning_rate=0.05, max_delta_step=0,
             max_depth=6, min_child_weight=1, missing=nan,
             monotone_constraints='()', n_estimators=1000, n_jobs=10,
             num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
             reg_lambda=1, scale_pos_weight=1, subsample=0.8,
             tree_method='exact', validate_parameters=1, verbosity=None)
```

```
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)
print_best_params(lgbm_reg, lgbm_params)
```

```
LGBMRegressor 5 CV 시 최적 평균 RMSE 값: 0.1163, 최적 alpha:{'n_estimators': 1000}
LGBMRegressor(colsample_bytree=0.4, learning_rate=0.05, n_estimators=1000,
              num_leaves=4, reg_lambda=10, subsample=0.6)
```

회귀 모델들의 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 모델 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: 0.10007930884470513
Ridge 모델의 RMSE: 0.10345177546603258
Lasso 모델의 RMSE: 0.10024170460890046

```
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: 0.10277689340617208
XGBM 모델의 RMSE: 0.10946473825650248
LGBM 모델의 RMSE: 0.10382510019327311

스태킹 앙상블 모델을 통한 회귀 예측

스태킹 모델에는 두 종류의 모델이 필요하다. (1) 개별적인 기반 모델 (2) 이 개별 기반 모델의 예측 데이터를 학습 데이터로 만들어 학습하는 **최종 메타 모델**
핵심은 여러 개별 모델의 예측 데이터를 각각 **스태킹 형태**로 결합해 최종 메타 모델의 학습용 피처 데이터셋과 피처 데이터셋을 만드는 것이다. 최종 메타 모델이 학습할 피처 데이터셋은 원본 학습 피처셋으로 학습한 개별 모델의 예측값을 스태킹 형태로 결합한 것이다. 이때 사용하는 함수는 `get_stacking_base_datasets()`

```
# 4장 분류에서 학습한 내용

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)
    # 추후 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
```

이제 get_stacking_base_datasets()를 모델별로 적용해 메타 모델이 사용할 학습 피처 데이터셋과 테스트 피처 데이터셋을 추출한다. 적용할 개별 모델: Ridge, Lasso, XGBoost, LightGBM.

```
# 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_te
lasso_train, lasso_test = get_stacking_base_datasets(lasso_reg, X_train_n, y_train_n, X_te
xgb_train, xgb_test = get_stacking_base_datasets(xgb_reg, X_train_n, y_train_n, X_test_n,
lgbm_train, lgbm_test = get_stacking_base_datasets(lgbm_reg, X_train_n, y_train_n, X_test_


# 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)

# 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: 0.09786044237371427

이 모델을 적용한 결과 테스트 데이터셋에서 여태까지 가장 좋은 성능 평가를 보여준다. 스태킹 모델이 분류뿐 아니라 회귀에서도 효과적으로 사용될 수 있다는 걸 알 수 있다.

정리

위의 두 예제를 실습하며 데이터 정제와 변환, 선형회귀/회귀 트리의 최적화를 통해 회귀 모델 성능 향상을 실습했다. 스태킹 모델을 적용하면 예측 성능이 향상되는 것도 알 수 있었다.

goblurry

이전 포스트
5. 회귀 (1)

0개의 댓글

댓글을 작성하세요

댓글 작성