

전기차 가격 예측 프로젝트 KUBIG

Team | Watt's the Price?

CONTENTS

01

주제 소개

- DACon 프로젝트 소개
- 변수 설명

02

EDA

- 데이터 분포 및 상관관계 분석
- 파생변수 생성
- 접근 방향 정리

03

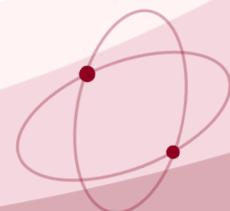
전처리

- 범주형 변수 인코딩
- 결측치 처리
- 이상치 처리
- 표준화

04

결과

- 모델링 1
- 모델링 2



1. 주제 소개

01. 주제 소개



대회 안내 데이터 코드 공유 토크 리더 보드 제출

[개요]

[배경]
안녕하세요 데이터 여러분! :)

전기차 가격 예측 해커톤: 데이터로 EV를 읽다에 오신 것을 환영합니다!

전기차 가격 예측은 빠르게 성장하는 전기차 시장에서 소비자와 제조사 모두에게 중요한 가치를 제공합니다.

정확한 가격 예측은 시장 경쟁력 분석, 소비자 구매 의사 결정 지원, 그리고 생산 및 유통 최적화에 기여할 수 있습니다.

이번 해커톤에서 전기차의 다양한 데이터를 바탕으로 가격을 예측하는 AI 알고리즘을 개발하는 것을 목표로 합니다.

여러분의 창의적인 아이디어와 데이터 분석 역량을 통해, 전기차 시장의 미래를 만들어 보세요!

[주제]
전기차와 관련된 데이터를 활용하여 전기차 가격을 예측하는 AI 알고리즘 개발

[설명]
전기차와 관련된 데이터를 활용하여 전기차 가격을 예측하는 AI 알고리즘 개발

⚡ 프로젝트 목표:

전기차의 주요 특성을 기반으로 가격을 예측하는 모델 개발

📊 데이터 개요:

- 전기차의 제조사, 모델, 차량 상태, 배터리 용량, 주행거리 등 다양한 변수 포함
- 가격 결정에 영향을 미치는 요인 분석

🔍 주요 연구 질문:

- 어떤 요인이 전기차 가격에 가장 큰 영향을 미치는가?
- 가격 예측 모델의 성능을 향상시키기 위한 최적의 접근법은 무엇인가?

01. 변수 설명

[변수 설명]

- ID: 차량별 고유 ID
- 제조사: 차량 제조사(H사, B사, K사, A사, T사, P사, V사 총 7개의 제조사 중 하나)
- 모델: 차량 모델명(총 21개의 모델 중 하나)
- 차량상태: Brand New(신차), Nearly New(준신차), Pre-Owned(중고차)
- 배터리용량: 잔존 배터리용량(kWh 단위)
- 구동방식: AWD(사륜구동), FWD(전륜구동), RWD(후륜구동)
- 주행거리(km): 누적 주행거리
- 보증기간(년): 무상 수리 및 서비스 기간
- 사고이력: No, Yes
- 연식(년): 0, 1, 2로 이루어짐(ex. 연식이 1: 출시된 지 1년 된 차량)
- 가격(백만원)



2. EDA

02-1. 데이터 brief view

`train.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7497 entries, 0 to 7496
Data columns (total 11 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   ID          7497 non-null    object  
 1   제조사       7497 non-null    object  
 2   모델         7497 non-null    object  
 3   차량상태     7497 non-null    object  
 4   배터리용량     4786 non-null    float64
 5   구동방식     7497 non-null    object  
 6   주행거리(km) 7497 non-null    int64  
 7   보증기간(년) 7497 non-null    int64  
 8   사고이력     7497 non-null    object  
 9   연식(년)     7497 non-null    int64  
 10  가격(백만원) 7497 non-null    float64
dtypes: float64(2), int64(3), object(6)
memory usage: 644.4+ KB
```

총 7497개의 행, 11개의 열로 구성된 데이터
feature: 범주형 변수 6개, 수치형 변수 4개
target: 가격(백만원) - 수치형 변수

[변수별 결측치 개수 및 비율]

ID	0
제조사	0
모델	0
차량상태	0
배터리용량	2711
구동방식	0
주행거리(km)	0
보증기간(년)	0
사고이력	0
연식(년)	0
가격(백만원)	0

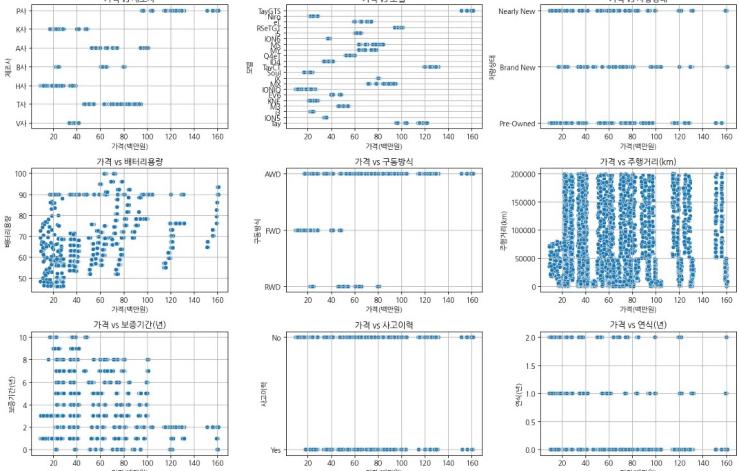
dtype: int64

배터리용량 null rate: 36.16%

‘**배터리용량**’ 변수에만

2711개(36.16%)의 결측값 존재

02-1. 데이터 분포 및 상관관계 분석



연속형 변수 간 상관관계

	배터리용량	주행거리(km)	보증기간(년)	연식(년)	가격(백만원)
배터리용량	1.00	-0.58	0.56	-0.02	0.43
주행거리(km)	-0.58	1.00	-0.66	-0.09	-0.04
보증기간(년)	0.56	-0.66	1.00	0.04	-0.35
연식(년)	-0.02	-0.09	0.04	1.00	-0.06
가격(백만원)	0.43	-0.04	-0.35	-0.06	1.00

◆ Insight

💡 가격에 가장 영향을 미치는 연속형 변수는 **배터리 용량**

💡 보증기간은 중고차 영향으로 인해 음의 상관관계를 가지는 것으로 보임

💡 연식과 주행거리는 가격과 직접적인 상관이 크지 않음

02-1. 시각화

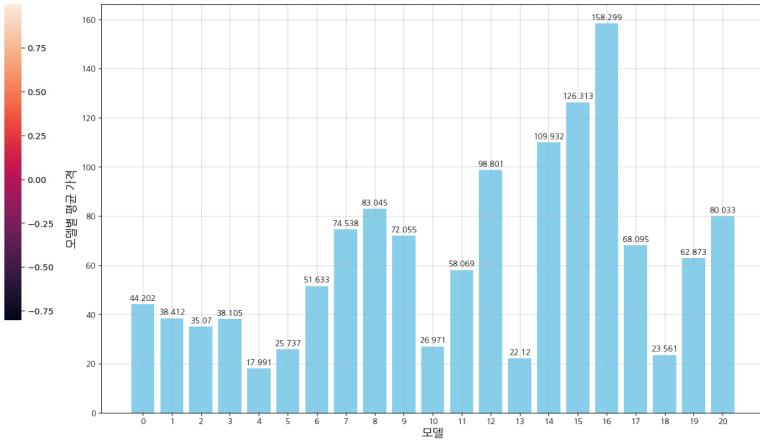
히트맵



가격(백만원)과 다른 변수들 간에 상관계수가 크지 않음

→ 선형성을 가정하는 일반 선형회귀, Ridge 회귀, Lasso 회귀보다는 **트리모델**을 사용하는 것이 낫겠다는 인사이트를 얻음.

모델 평균 가격 막대그래프

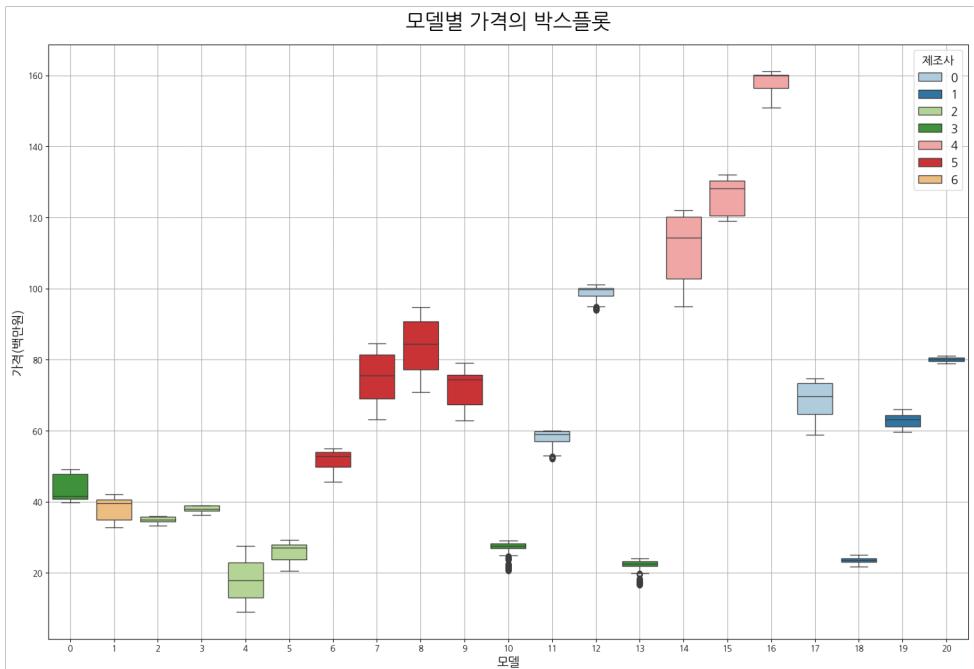


모델별로 모델평균가격의 편차가 큼을 확인
→ '모델평균가격'을 하나의 열로 넣는 것이 타당함

02-2. 파생변수 생성

① 모델평균가격

```
mean_price = train.groupby(['모델'])['가격(백만원)'].mean()  
train['모델평균가격'] = train['모델'].map(mean_price)
```



같은 제조사의 차량이더라도 가격 차이가
큽

& 같은 모델끼리는 가격의 편차가 작아짐

💡 동일한 모델이라면 가격이 유사함!

‘모델평균가격’ 파생변수 생성

02-2. 파생변수 생성

② 연식 대비 주행거리(mileage_per_year)

- ◆ 차량 사용자마다 **차량 사용 강도**가 다름 → 이는 단순 연식과 주행거리 값만으로는 **판단 불가**
- ◆ 차량 사용 강도를 통해 **감가율** 평가 가능
- ◆ 연식 대비 주행거리가 높다면 배터리 충전 및 방전이 빈번했을 것 → **배터리 상태 bad**
→ 배터리 수명은 가격에 치명적 영향

➡ 주행거리(km) / 연식(년)으로 계산한 'mileage_per_year' 파생변수 생성

```
# 연식 = 0이면 그냥 주행거리(km) 값 그대로 사용
train['mileage_per_year'] = train.apply(
    lambda row: row['주행거리(km)'] / row['연식(년)'] if row['연식(년)'] != 0 else row['주행거리(km)'],
    axis=1)
```

02-3. 표준화

✓ 표준화: StandardScaler vs MinMaxScaler

- ◆ Right-Skewed된 변수('주행거리(km)')가 있어 최대/최소값에 민감한 MinMaxScaler는 부적절
- ◆ StandardScaler는 데이터가 정규분포를 따르지 않더라도 변수의 분포는 유지하므로 안정적
- ◆ 범주형 변수가 많은데, 수치형 변수를 이와 같은 범위로 표준화하는 것(MinMaxScaler: 0~1)이 부적절

➡ **StandardScaler**를 이용해서 표준화

02. EDA 이후 접근 방향 정리

📌 접근 방향 정리

1) 전처리

🔋 배터리용량 결측치를 어떻게 처리할 것인가? (단순평균값 / Ridge 회귀 등을 통해 다른 변수와의 관계 이용)

⚖️ standard / minmax scaling 선택

❗ 이상치를 처리할 것인가 (O/X)

2) 모델 적합

🌳 트리 기반, 회귀, 딥러닝 모델 중 어느 모델?

⚙️ 하이퍼파라미터 튜닝 (GridSearch / RandomSearch / Bayesian)



03. 전처리 - 모델링 1

03-1. 범주형 변수 라벨 인코딩

✓ LabelEncoder vs One-Hot Encoding

- ◆ 범주형 변수의 범주 개수가 매우 많음 → One-Hot Encoding 사용 시 차원이 커질 위험 존재
 - ◆ LabelEncoder()는 변수에 잘못된 순서를 부여할 수 있다는 단점 존재
- ↔ but 트리 기반 모델은 숫자의 순서를 직접적으로 활용하지 않기에, 해당 문제점 해결 가능

```
# LabelEncoder 적용
for col in train:
    if (train[col].dtype == 'object') & (col != 'ID'):
        le = LabelEncoder()
        train[col] = le.fit_transform(train[col])
```

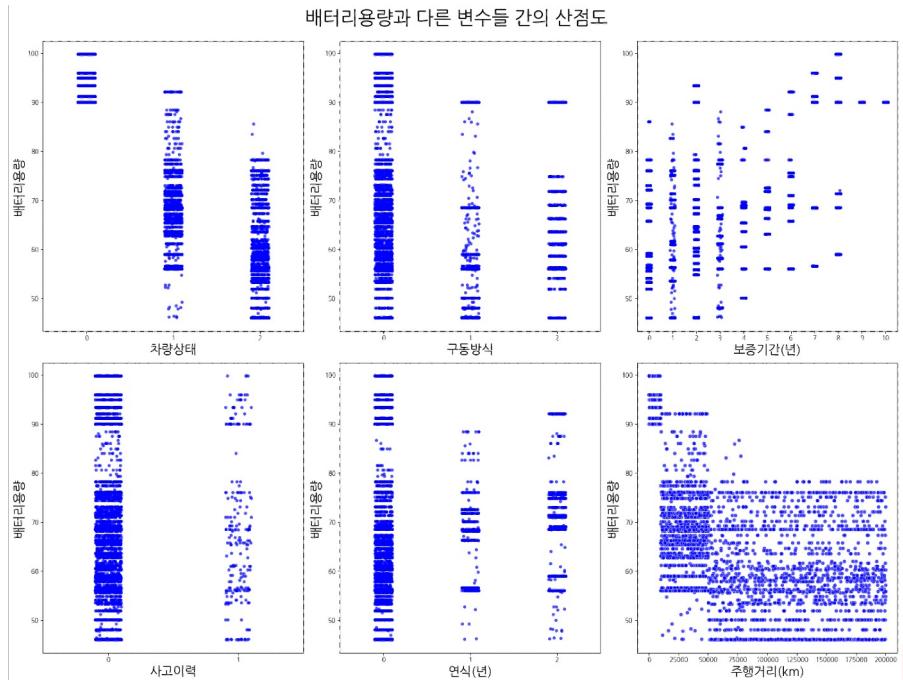
➡ LabelEncoder 사용 !

제조사 매핑: {0: 'AA사', 1: 'B사', 2: 'H사', 3: 'K사', 4: 'P사', 5: 'T사', 6: 'V사'}
모델 매핑: {0: 'EV6', 1: 'ID4', 2: 'ION5', 3: 'ION6', 4: 'IONIQ', 5: 'KNE', 6: 'M3',
7: 'MS', 8: 'MX', 9: 'MY', 10: 'Niro', 11: 'Q4eT', 12: 'RSeTGT', 13: 'Soul',
14: 'Tay', 15: 'TayCT', 16: 'TayGTS', 17: 'eT', 18: 'i3', 19: 'i5', 20: 'iX'}
차량상태 매핑: {0: 'Brand New', 1: 'Nearly New', 2: 'Pre-Owned'}
구동방식 매핑: {0: 'AWD', 1: 'FWD', 2: 'RWD'}
사고이력 매핑: {0: 'No', 1: 'Yes'}

➡ 매핑 결과

03-2. 결측치 처리

✓ 배터리용량과 다른 변수들 간 관계 확인



- 1 배터리용량 & 차량상태 상관계수
 $= -0.7949794823254366$
- 2 배터리용량 & 주행거리(km) 상관계수
 $= -0.5765384147171888$

◆ 모델별로 차량 제조 단계부터
배터리용량에 차이가 존재



‘모델별 배터리용량의 평균값’도 배터리용량
결측치 예측에 활용

03-2. 결측치 처리

- ✓ 차량상태, 주행거리(km), 모델별 배터리용량의 평균값을 독립변수(X), 배터리용량을 종속변수(Y)로 하는 **Ridge 모델** 적합

💡 RandomForest vs Ridge

- ◆ 두 변수 간의 선형 관계를 측정하는 **상관계수가** 모든 변수 쌍에서 **높은 값을** 보임 → **선형 관계** 존재
- ◆ RandomForest도 이용해 본 결과, Ridge를 사용해서 결측값을 대체했을 때 모델 성능이 더 뛰어남
- ◆ Ridge는 이상치에 민감한 모델 → 따라서 StandardScaler를 이용한 **표준화** 후 모형 적합

- ✓ RandomizedSearchCV를 활용하여 최적의 하이퍼파라미터 탐색

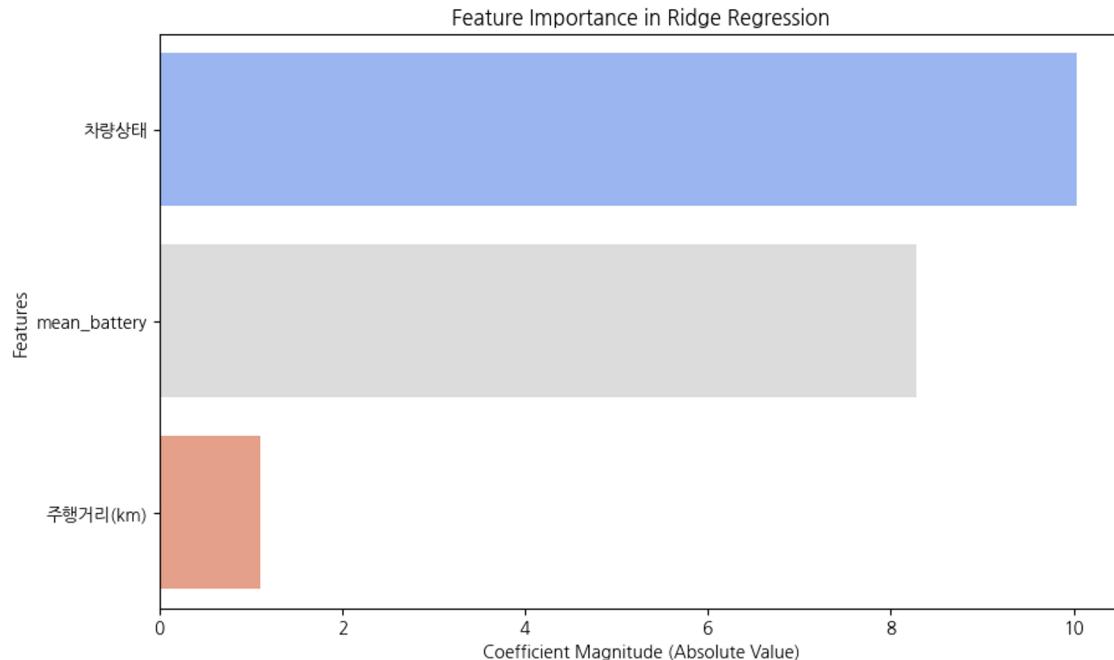
```
# 하이퍼파라미터 그리드
param_grid = {
    'alpha' : [0.0001, 0.001, 0.01, 0.1, 1, 10, 100],
    'fit_intercept' : [True, False],
    'solver': ['auto', 'svd', 'cholesky', 'lsqr', 'saga']
}
# scoring 방식은 RMSE
ridge_cv_model = RandomizedSearchCV(ridge, param_grid, n_iter = 100,
                                      scoring = 'neg_root_mean_squared_error', cv = 5)
ridge_cv_model.fit(X, y)
```

➡ 선택된 하이퍼파라미터

'solver' = saga
'fit_intercept' = True
'alpha' = 0.01

03-2. 결측치 처리

Feature Importance 시각화 - 💡 회귀계수의 절댓값 활용



4. 결과 - 모델링 1

04-1. 변수 선택

'ID'는 차량별 고유번호일 뿐이며, '제조사'의 정보는 '모델'의 정보로 대체 가능하므로 drop

다중공선성 유발 변수 제거

	VIF Factor	features
0	5.359951	모델
1	9.546464	차량상태
2	47.123593	배터리용량
3	1.822127	구동방식
4	713.513805	주행거리(km)
5	16.857503	보증기간(년)
6	1.047678	사고이력
7	4.017979	연식(년)
8	13.995905	모델평균가격
9	720.042352	mileage_per_year

→
파생변수 제외
VIF가 가장 큰
'주행거리(km)' 제거

	VIF Factor	features
0	5.352891	모델
1	9.464850	차량상태
2	46.855404	배터리용량
3	1.821480	구동방식
4	16.730928	보증기간(년)
5	1.047032	사고이력
6	1.542628	연식(년)
7	13.958493	모델평균가격
8	6.918720	mileage_per_year

→
다중공선성 완화됨
(배터리용량의 결측값을
피처들을 활용해 대체하
였으므로 VIF가 높은 것
은 자연스러움)

↓
'주행거리(km)' drop

04-2. 모델 적합

✓ 표준화: StandardScaler vs MinMaxScaler

- ◆ Right-Skewed된 변수('주행거리(km)')가 있어 최대/최소값에 민감한 MinMaxScaler는 부적절
- ◆ StandardScaler는 데이터가 정규분포를 따르지 않더라도 변수의 분포는 유지하므로 안정적
- ◆ 범주형 변수가 많은데, 수치형 변수를 이와 같은 범위로 표준화하는 것(MinMaxScaler: 0~1)이 부적절

➡ **StandardScaler**를 이용해서 표준화

04-2. 모델 적합

✓ 다양한 머신러닝 모델 적합 시도:

Lasso 회귀 / RandomForest Regressor / LightGBM / XGBoost 등

➡ **RandomForest Regressor**를 이용한 모델이 가장 높은 성능을 보임

- ◆ 변수들 간의 관계가 **비선형적** → 선형 회귀보다는 **트리 모델들의** 성능이 우수
- ◆ **부스팅 모델** → **과적합 위험**이 있음

실제로, 랜덤 포레스트 모델보다 RMSE는 미세하게 낮지만, test set에 적합했을 때 성능은 떨어짐

- ◆ **랜덤 포레스트는** 정규분포를 따르지 않는 데이터와 이상치에 **강한 반면**, **부스팅 모델은** 이상치에 **민감함**
- ◆ 부스팅 모델보다 랜덤 포레스트 모델의 성능이 좋았던 것을, 주어진 데이터셋 내부에

복잡한 패턴이 많지 않았으며, 이러한 **단순한 구조를** 부스팅 모델이 **오버피팅**했다고 해석할 수 있음

04-2. 모델 적합

✓ RandomizedSearchCV를 이용한 하이퍼파라미터 탐색

```
rf = RandomForestRegressor()
rf_params = {
    'n_estimators' : [50, 100, 200],
    'max_depth' : [10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5],
}
rf_cv_model = RandomizedSearchCV(rf, rf_params, cv = 5,
    scoring = 'neg_root_mean_squared_error')
rf_cv_model.fit(X_train, y_train)
```

```
best n_estimators: 200
best max_depth: 10
best min_samples_split: 5
best min_samples_leaf: 5
```

```
-----
Model Performance:
MSE: 1.6629569983415151
MAE: 0.6703213801499274
RMSE: 1.2895569000015141
R2 Score: 0.998854195020023
```

➡ 선택된 하이퍼파라미터: n_estimators = 200, max_depth = 10,

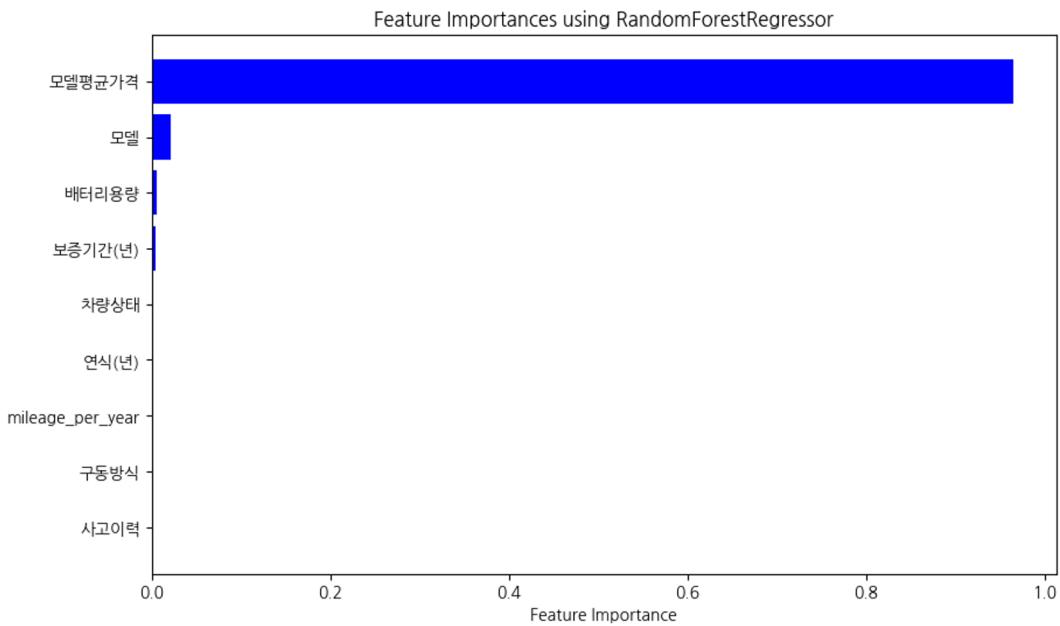
min_samples_split = 5, min_samples_leaf = 5

➡ validation set에 대한 RMSE = 1.2895569

➡ Public Score = 1.0170744974, Private Score = 1.2206657563

04-2. 모델 적합

✓ Feature Importance 시각화

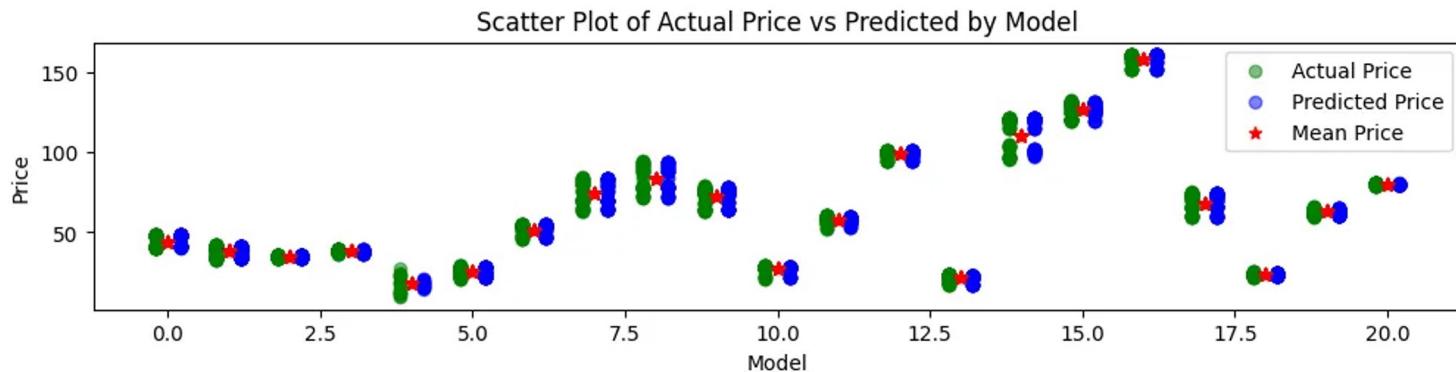


모델평균가격, 모델,
배터리용량, 보증기간(년)이
중요한 변수

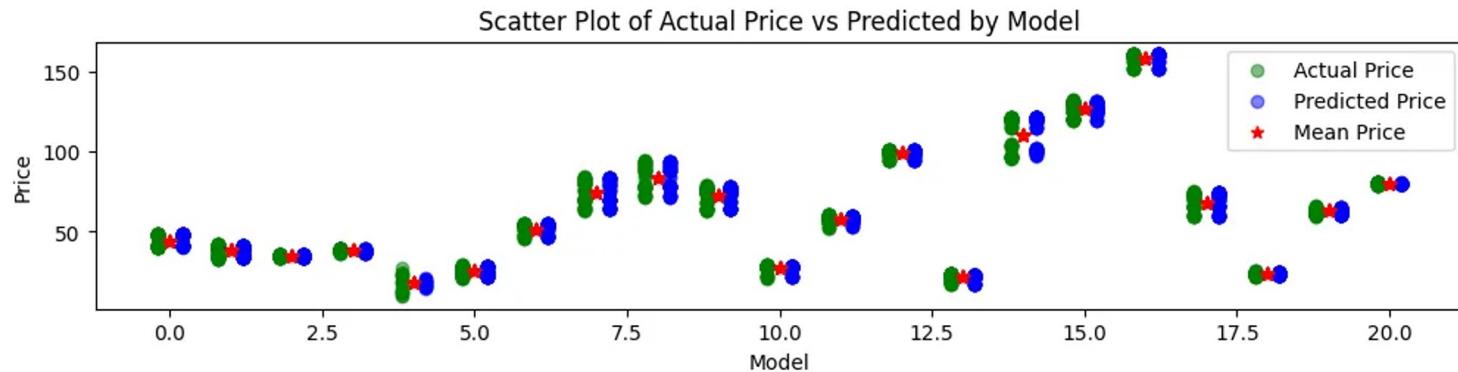
3. 전처리 - 모델링 2

03. 모델링1의 결과 분석

- ✓ 모델링1의 지속적인 하이퍼파라미터 튜닝에도 private score 1.22대에 교착
 - ➡ 모델링 1의 F1이 높은 **모델 평균가격이 성능 고착화에 영향을 줄 것으로 예상**
 - ➡ 각 차량모델별로 예측 가격, 실제 가격, 모델 평균가격을 plot하여 살펴보자.



03. 모델링1의 결과 분석



- ✓ 각 차량 모델별로 예측가격과 실제가격 간에 분산차이가 있을 것으로 예상됨.
- ➡ One way-ANOVA F 검정 결과 통계적 유의성이 없음.
- ➡ 분산은 잘 맞추고 있으나 RMSE가 문제

03. 모델링1의 결과 분석

Model 0: RMSE = 0.6513

Model 1: RMSE = 0.6093

Model 2: RMSE = 0.3296

Model 3: RMSE = 0.2888

Model 4: RMSE = 5.8209

Model 5: RMSE = 0.6038

Model 6: RMSE = 0.6056

Model 7: RMSE = 0.6969

Model 8: RMSE = 0.6571

Model 9: RMSE = 0.8459

Model 10: RMSE = 0.5663

Model 11: RMSE = 0.4506

Model 12: RMSE = 0.3150

Model 13: RMSE = 0.6061

Model 14: RMSE = 2.5441

Model 15: RMSE = 3.7554

Model 16: RMSE = 0.2836

Model 17: RMSE = 0.4578

Model 18: RMSE = 0.3207

Model 19: RMSE = 0.5652

Model 20: RMSE = 0.5604

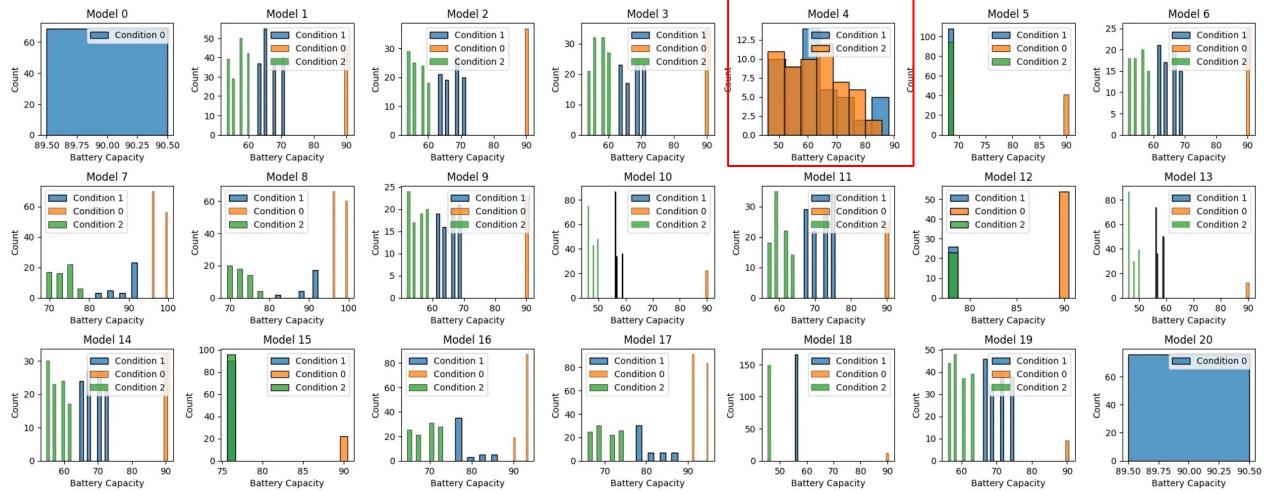
각 차량모델별 예측가격의 분산은 문제가 없음

차량 모델 4번의 RMSE: 5.8209

차량 모델 14번, 15번의 RMSE는 1을 초과

➡ 모델링 1의 예측은 잘 이루어지나 특정 차량 모델의 데
이터가 모델링 1의 예측 방식과 잘 맞지 않는다.

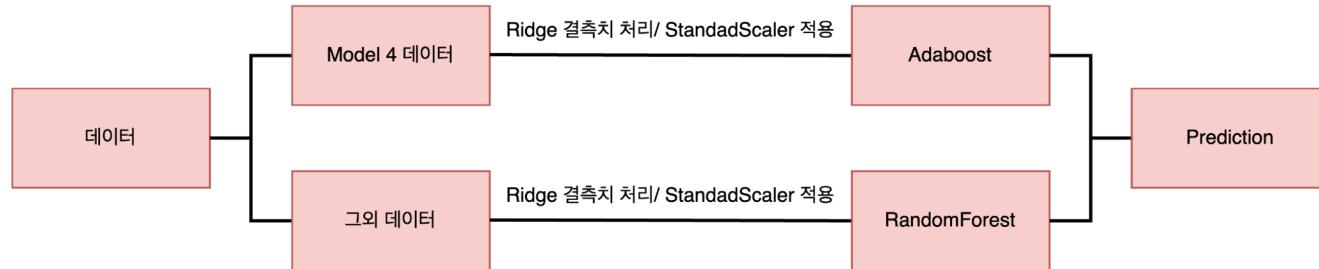
03. 모델링1의 결과분석



➡ 4번 차량모델 외 차량모델은 각 상태마다 그룹을 이루고 있음

➡ 4번 차량모델의 데이터가 나머지 차량모델의 예측에 방해

03. 모델링2 컨셉



- 💡 Train, Test data set을 차량 모델 4와 그외 차량 모델들로 분리
- 💡 결측치 **Ridge** 학습 및 예측으로 결측치 처리
- 💡 **StandardScaler** 적용
- 💡 모델 4를 제외한 차량 모델들에 대하여 **RandomForest Regressor** 학습 및 예측
- 💡 차량 모델 4에 대하여 **Adaboost** 학습 및 예측

03. 전처리 - 최종 변수 선택

```
# model 4 insurance period 8 drop
train = train[~((train['model'] == 4) & (train['insurance_period'] == 8))]

# over 50000km newly new drop
train = train[~((train['condition']== 1)&(train['mileage']>50000))]

# mileage per year column drop
train = train.drop('mileage_per_year', axis = 1)

# year column drop
train = train.drop('year', axis = 1)

# drive type column drop
train = train.drop('drive_type', axis = 1)
```

- ✓ 모델 4 보증기간 == 8년 컬럼 제거
- ✓ 차량상태==1 & 연비>50000 제거
- ✓ 파생변수 mileage_per_year 제거
- ✓ 연식(년) 변수 제거
- ✓ 구동방식 변수 제거

파생변수: 모델 1과 달리, 모델평균가격만 적용

03. 전처리 - 결측치 처리 방안

- ✓ 대부분의 모델과 다른 경향성을 가진 차량모델 4번이 결측치 예측에 방해
- ➡ 차량모델 4번을 분리하여 처리하자.
- ➡ 차량모델 4번의 결측치 처리를 평균 대입, RandomForest, Ridge 예측 방식을 검토
- ➡ **Ridge**로 결측치를 처리했을 때 Price 예측 성능이 높았다.

모델링 1과 동일하게, 차량상태, 주행거리(km), 모델별 배터리용량의 평균값 의 3가지 변수로 Ridge 회귀 적용

4. 결과 - 모델 2

04-2. 모델 적합 (차량 모델 4 제외한 나머지 차량 모델들)

RandomForest Regressor 학습 / 예측

하이퍼파라미터 최적화 X (GridSearchCV 효과 없었음)

```
# train data set으로 학습 및 훈련 rmse
# model 4 제외
train7_others = train7[train7['model']!=4]
X_ridge = train7_others.drop(columns = ['ID', 'price'])
y_ridge = train7_others['price']
X_train_ridge, X_test_ridge, y_train_ridge, y_test_ridge_others = train_test_split(X_ridge, y_ridge, test_size = 0.2, random_state = 42)
# 학습
ridge_null_model_rf_others = RandomForestRegressor(n_estimators = 100, max_depth = 10, min_samples_split = 5, min_samples_leaf = 2, random_state = 17)

# scaling
scaler0 = StandardScaler()
X_train_ridge_scaled = scaler0.fit_transform(X_train_ridge)
X_test_ridge_scaled = scaler0.transform(X_test_ridge)

ridge_null_model_rf_others.fit(X_train_ridge_scaled, y_train_ridge)
```

others RMSE: 1.0631546197261716
model 4 RMSE: 5.942567197273071
total RMSE: 1.278152476947745

▶ 선택된 하이퍼파라미터: n_estimators = 100, max_depth = 10, min_samples_split = 5, min_samples_leaf = 2, random_state=17

▶ validation set에 대한 RMSE = 1.284884713401

▶ Public Score = 0.9213119795, Private Score = 1.2055857718

04-2. 모델 적합 - 차량 모델 4 (Adaboost)

왜 모델 4에 대해서는 **Adaboost** 학습 / 예측 ?

- ◆ 차량 모델 4번에 대한 EDA를 통해 확인해본 결과, K-means 등을 수행하며 가격대의 예측이 각 독립변수와 큰 연관성이 없었다.
- ◆ 같은 가격이라도 각각의 독립변수들이 같지는 않았다.
- ◆ 따라서 train 데이터의 차량모델4번의 이러한 분포가 test에서도 동일할 것이라는 가정에서 각 데이터 포인트의 가격을 정밀하게 맞추는 것이 더 중요할 것 같다.
- ➡ 따라서 맞추지 못한 값에 가중치를 주는 **Adaboost** 방식으로 접근하면 비교적 RMSE를 낮추기에 적합할 것으로 판단

04-2. 모델 적합 - 차량 모델 4 (Adaboost)

✓ 하이퍼파라미터 최적화 X (GridSearchCV 효과 없었음)

```
# model 4
train7_model4 = train7[train7['model'] ==4]
X_ridge = train7_model4.drop(columns = ['ID', 'price'])
y_ridge = train7_model4['price']
X_train_ridge, X_test_ridge1, y_train_ridge, y_test_ridge_4 = train_test_split(X_ridge, y_ridge, test_size = 0.2, random_state = 42)
#학습
#학습 모델
ridge_null_model_rf_4 = RandomForestRegressor(n_estimators = 100, max_depth = 10,min_samples_split = 5, min_samples_leaf = 2 )
ridge_null_model_gbm_4= GradientBoostingRegressor(n_estimators = 100, learning_rate = 0.01, max_depth = 10,min_samples_leaf = 2)
ridge_null_model_ada_4= AdaBoostRegressor(n_estimators=200, learning_rate=0.01,estimator= DecisionTreeRegressor(max_depth =3), random_state=42)
#scaling
scaler1 = StandardScaler()
X_train_ridge_scaled = scaler1.fit_transform(X_train_ridge)
X_test_ridge_scaled = scaler1.transform(X_test_ridge1)
#fit
# ridge_null_model_rf_4.fit(X_train_ridge_scaled, y_train_ridge)
# ridge_null_model_gbm_4.fit(X_train_ridge_scaled, y_train_ridge)
ridge_null_model_ada_4.fit(X_train_ridge_scaled, y_train_ridge)
```

others RMSE: 1.0631546197261716
model 4 RMSE: 5.942567197273071
total RMSE: 1.278152476947745

→ 선택된 하이퍼파라미터: n_estimators=200, learning_rate=0.01,
estimator= DecisionTreeRegressor(max_depth =3), random_state=42

→ validation set에 대한 RMSE = 1.284884713401

→ Public Score = 0.9213119795, Private Score = 1.2055857718

04-2. 최종 모델 적합 (차량 모델 4 + 나머지 적합)

```
# train data set 전체 합성
y_pred_total = np.concatenate([y_pred_ridge_others,y_pred_ridge_4 ])
y_test_total = np.concatenate([y_test_ridge_others, y_test_ridge_4])
mse_ridge = mean_squared_error(y_test_total, y_pred_total)
rmse_ridge = np.sqrt(mse_ridge)
print('total RMSE:', rmse_ridge)
x_test_total = pd.concat([X_test_ridge, X_test_ridge1])

# test data도 동일하게 진행
test7_others = test7[test7['model']!=4]
test7_model4 = test7[test7['model'] ==4]
    #others
test7_others_scaled = scaler0.transform(test7_others.drop('ID',axis =1))
test7_others_predict = ridge_null_model_rf_others.predict(test7_others_scaled)
#test7_others_predict = best_model_others.predict(test7_others_scaled)
test7_others['predictions']= test7_others_predict
    #model4
test7_model4_scaled = scaler1.transform(test7_model4.drop('ID',axis =1))
# test7_4_predict = ridge_null_model_rf_4.predict(test7_model4_scaled)
# test7_4_predict = ridge_null_model_gbm_4.predict(test7_model4_scaled)
test7_4_predict = ridge_null_model_ada_4.predict(test7_model4_scaled)
# test7_4_predict = best_model.predict(test7_model4_scaled)

test7_model4['predictions'] =test7_4_predict

test7_predict = pd.concat([test7_others, test7_model4])

test7_predict.sort_values(by= 'ID', inplace= True)
```

4. 결과정리

04. 모델 적합 기록

모델 적합 기록

표 +

Aa 이름	모델 종류	사용한 하이퍼파라미터 딕셔너리	# 훈련(검증)셋 RMSE	# Public Score	# Private Score	비고
정제	LGBM	'n_estimators': [100, 200], 'learning_rate': [0.01, 0.1, 0.2], 'num_leaves': [10, 20, 30], 'min_data_in_leaf': [40, 50, 60]	1.1881	1.0339282392	1.2504194043	- 베타리uing 평균값으로 처리 - 이상치 그대로 - 변수 다 그대로 - standard scaling 진행
정제	RandomForestRegressor	'n_estimators': [100, 200], 'max_depth': [None, 10, 20], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]	1.0674	0.8974733232	1.1612660774	- 전처리 위와 동일 - GridSearchCV - cv=5
수환 basic RF	RandomForestRegressor	(n_estimators=100, random_state=42)		1.2111645955	1.3332296734	
채영	RandomForestRegressor	(n_estimators = 50, max_depth = 10, min_samples_split = 10, min_samples_leaf = 5)	1.344066365756	1.0493093806	1.2377017205	- GridSearchCV 사용 - 다른공선성을 높이는 '주행거리(km)' 변수 제거
채영	XGBoost	{n_estimators = 200, learning_rate = 0.05, max_depth = 5, min_child_weight = 3, gamma = 0, alpha = 0.1, lambda = 0.1}	1.301382882176	1.0478453847	1.2495356763	- GridSearchCV 사용
수환 RF RF	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.320645024865	1.2252366482	1.6646784607	Rf 방식으로 결측치 예측 이때 mean battery 값에 condition 조건 포함/ RF로 가격 추정
수환 wise RF	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.323348684643	1.0823668491	1.3866430146	차량 model, condition wise mean으로 결측치 처리 / RF로 가격 추정
정제	XGBoost	"n_estimators": [100, 200], "learning_rate": [0.01, 0.05, 0.1], "max_depth": [3, 5, 7], "min_child_weight": [1, 3, 5]		1.3791992641	1.4840064572	
수환 wise RF	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split =	1.301280726486	1.1466560908	1.2924376028	average price를 추가

04. 모델 적합 기록

정체	XGBoost	"n_estimators": [100, 200], "learning_rate": [0.01, 0.05, 0.1], "max_depth": [3, 5, 7], "min_child_weight": [1, 3, 5]		1.3791992641	1.4840064572	
수환 wise RF average price	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.301280726486	1.1466560908	1.2924376028	average price를 추가
▣ 채영	RandomForestRegressor	{n_estimators = 50, max_depth = 10, min_samples_split = 10, min_samples_leaf = 5}	1.312755857018	1.0423967783	1.2203471648	배터리용량 결측값 채울 때 RandomForest 대신 Ridge 회귀 모델 적합
▣ 수환 wise RF average scaled	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.302530170929	1.1309982271	1.303147398	min max scaler 사용
채영	Lasso 회귀	{alpha = 0.01, fit_intercept = True}	3.749952472808	3.6498082734	3.7860934419	회귀는 아무래도... 아닌게 맞는 듯 하네요
▣ 수환 wise RF average scaled-best	RandomForestRegressor	{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200}	1.3464	1.0699710676	1.285517732	hyperparameter tuning 후의 결과
채영	RandomForestRegressor	{n_estimators = 100, max_depth = 10, min_samples_split = 10, min_samples_leaf = 5}	1.301970403244	1.0332120519	1.2113725241	StandardScaler로 표준화, Ridge로 배터리용량 결측값 채움
▣ 수환 ridge RF average price	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.320531800407	1.0484644854	1.2616307613	ridge alpha=0.7로 결측치 채움, 그 후에 average price 추가하고 RF로 예측
▣ 수환 ridge RF average scaled	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.302030856453	1.0394427346	1.254402255	min max scaled 사용
▣ 수환 ridge RF average scaled	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.301527466037	1.0167572522	1.249340363	Ridge alpha 값을 1.0으로 변경함/ 성능 향상이 보임 minmax scaling

04. 모델 적합 기록

수한 model4 split	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.297851807210	0.9516387628	1.2308735107	model4를 전체와 분리함. 나머지에 대하여 채영과 비슷한 방식으로 진행. model4에 대해 평균으로 결측치 처리하고 RF 적용 현재 train 셋에서 model4에 대한 rmse가 높기에 model4에 대한 학습을 고도화하면 전반적인 score 향상이 가능할 듯 싶음
수한 model4 split baseline	RandomForestRegressor	n_estimators = 100, max_depth = 10,min_samples_split = 10, min_samples_leaf = 5	1.292774061879	0.9469037035	1.2209789359	baseline에 standard scaler 적용(채영, 정재 제안) 다음 모델은 model4만의 RMSE를 개선하는 것으로 방향을 정함.
수한 model4 split-adaboost	RandomForestRegressor &Adaboost	rf: {n_estimators = 100, max_depth = 10,min_samples_split = 5, min_samples_leaf = 2} ada: {n_estimators=100, learning_rate=0.01,estimator=DecisionTreeRegressor(max_depth =3), random_state=42}	1.288224529300	0.929254965	1.2085687794	model4 에는 adaboost(decisiontree를 week learner로)
수한 model4 split-adaboost hyper	RandomForestRegressor &Adaboost	rf: {n_estimators = 100, max_depth = 10,min_samples_split = 5, min_samples_leaf = 2} ada: {n_estimators=200, learning_rate=0.01,estimator=DecisionTreeRegressor(max_depth =3), random_state=42}	1.286822405066	0.9026097678	1.2012232206	
수한 model4-RF fix	RandomForestRegressor &Adaboost	rf: {n_estimators = 100, max_depth = 10,min_samples_split = 5, min_samples_leaf = 2,random_state=17} ada: {n_estimators=200, learning_rate=0.01,estimator=DecisionTreeRegressor(max_depth =3), random_state=42}	1.284884713401	0.9213119795	1.2055857718	others RMSE: 1.0632292848352816 model 4 RMSE: 6.039474434831992 total RMSE:



Thank You