# 9장 랜덤 포레스트 : 자동차 데이터셋

#### 학습 목표

결정 트리의 발전된 형태인 기초인 랜덤 포레스트를 학습하고, 교차검증 방법을 활용하여 모델을 평가해 더 나은 모델을 찾아봅시다.

#### 학습 순서



#### 랜덤 포레스트 소개

랜덤 포레스트Random Forest 모델은 결정 트리의 단점인 오버피팅 문제를 완화시켜주는 발전된 형태의 트리 모델입니다. 랜덤으로 생성된 무수히 많은 트리를 이용하여 예측을 하기 때문에 랜덤 포레스트라 불립니다. 이렇게 여러 모델(여기서는 결정 트리)을 활용하여 하나의 모델을 이루는 기법을 앙상블이라 부릅니다.

<용어/>

**앙상블 기법**

여러 모델을 만들고 각 예측값들을 투표/평균 등으로 통합하여 더 정확한 예측을 도모하는 방법

</>



#### 장단점

| **장점** | **단점** |
| --- | --- |
| 결정 트리와 마찬가지로, 아웃라이어에 거의 영향을 받지 않습니다. | 학습 속도가 상대적으로 느린 편입니다. |
| 선형/비선형 데이터에 상관없이 잘 작동합니다. | 수많은 트리를 동원하기 때문에 모델에 대한 해석이 어렵습니다. |

#### 유용한 곳

* 종속변수가 연속형 데이터와 범주형 데이터인 경우 모두에서 사용할 수 있습니다.
* 아웃라이어가 문제가 되는 경우 선형 모델보다 좋은 대안이 될 수 있습니다.
* 오버피팅 문제로 결정 트리를 사용하기 어려울 때, 랜덤 포레스트를 사용할 수 있습니다.

#### TOP 10 선정 이유

* 앙상블 기법을 사용한 트리 기반 모델 중 가장 보편적인 방법입니다. 이후에 다루게 될 부스팅 모델에 비하면 예측력이나 속도에서 부족한 부분이 있고, 시각화 부분에서는 결정 트리에 못미치나, 다음 단계인 부스팅 모델을 이해하려면 꼭 알아야 할 필수 알고리즘입니다.

## 9.1 문제 정의 : 한눈에 보는 분석 목표

<금토끼의 문제 정의> 금토끼는 얼마 전에 새 자동차를 구입했습니다. 기존에 타던 자동차를 중고로 판매하려고 하는데 중고차 판매가 처음이라 적정 가격을 책정하기가 난감했습니다. 그래서 금토끼는 기존 중고차 거래의 이력이 담긴 자동차 모델명, 연식, 마일리지, 기타 자동차 성능 데이터를 사용하여, 중고차 가격을 예측하는 모델을 만들어보기로 했습니다.

| **난이도** | ⭐⭐☆ | | |
| --- | --- | --- | --- |
| **알고리즘** | 랜덤 포레스트(Random Forest) | | |
| **데이터셋 파일명** | car.csv | **종속 변수** | selling\_price(판매 가격) |
| **데이터셋 소개** | 중고차 판매 이력을 다룬 데이터입니다. 종속 변수는 판매가격이며, 독립변수로는 생산년도, 주행거리, 변속기, 마일리지, 배기량 등이 있습니다. | | |
| **문제 유형** | 회귀 | **평가지표** | RMSE |
| **사용한 모델** | RandomForestRegressor | | |
| **사용 라이브러리** | * numpy (numpy==1.19.5) * pandas (pandas==1.3.5) * seaborn (seaborn==0.11.2) * matplotlib (matplotlib==3.2.2) * sklearn (scikit-learn==1.0.2) | | |
| **예제 코드 노트북** | 위치 : https://github.com/musthave-ML10/notebooks/blob/main/  파일 : 09\_Random Forest.ipynb | | |

## 9.2 라이브러리 및 데이터 불러오기, 데이터 확인하기

4개의 필수 모듈과 car.csv 파일을 불러오겠습니다.

| import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns  file\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/car.csv' data = pd.read\_csv(file\_url) # 데이터셋 읽기 |
| --- |

그럼 head() 함수로 데이터의 전반적인 모습을 보겠습니다.

| data.head() # 상위 5행 출력 |
| --- |

|  | **name** | **year** | **selling\_price** | **km\_driven** | **fuel** | **seller\_type** | **transmission** | **owner** | **mileage** | **engine** | **max\_power** | **torque** | **seats** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Maruti Swift Dzire VDI | 2014 | 450000 | 145500 | Diesel | Individual | Manual | First Owner | 23.4 kmpl | 1248 CC | 74 bhp | 190Nm@ 2000rpm | 5.0 |
| **1** | Skoda Rapid 1.5 TDI Ambition | 2014 | 370000 | 120000 | Diesel | Individual | Manual | Second Owner | 21.14 kmpl | 1498 CC | 103.52 bhp | 250Nm@ 1500-2500rpm | 5.0 |
| **2** | Honda City 2017-2020 EXi | 2006 | 158000 | 140000 | Petrol | Individual | Manual | Third Owner | 17.7 kmpl | 1497 CC | 78 bhp | 12.7@ 2,700(kgm@ rpm) | 5.0 |
| **3** | Hyundai i20 Sportz Diesel | 2010 | 225000 | 127000 | Diesel | Individual | Manual | First Owner | 23.0 kmpl | 1396 CC | 90 bhp | 22.4 kgm at 1750-2750rpm | 5.0 |
| **4** | Maruti Swift VXI BSIII | 2007 | 130000 | 120000 | Petrol | Individual | Manual | First Owner | 16.1 kmpl | 1298 CC | 88.2 bhp | 11.5@ 4,500(kgm@ rpm) | 5.0 |



다음은 info() 함수로 컬럼별 형태를 보겠습니다.

| data.info() # 변수 특징 출력 |
| --- |



❶ 몇 개 컬럼에서 결측치가 있습니다. head() 정보와 비교하면 숫자 형태로 인식되어야 할 데이터가 object형으로 되어 있기도 합니다. 예를 들어 ❷ engine 변수는 숫자형 데이터여야 처리가 쉬운데 데이터 뒤에 ‘cc’가 붙어서 문자형 데이터로 받아들여집니다. 이런 부분은 데이터 클리닝 과정에서 숫자 형태로 변경합니다. 출력문에서 이와 같은 부분이 있는지 꼼꼼이 확인하고 넘어가야 합니다.

마지막으로 describe()로 통계적 정보를 확인합시다. 소수점 둘째 자리에서 반올림하겠습니다.

| round(data.describe(), 2) # 통계 정보 출력 |
| --- |

|  | **year** | **selling\_price** | **km\_driven** | **seats** |
| --- | --- | --- | --- | --- |
| **count** | 8128.00 | 8128.00 | 8128.00 | 7907.00 |
| **mean** | 2013.80 | 638271.81 | 69819.51 | 5.42 |
| **std** | 4.04 | 806253.40 | 56550.55 | 0.96 |
| **min** | 1983.00 | 29999.00 | 1.00 | 2.00 |
| **25%** | 2011.00 | 254999.00 | 35000.00 | 5.00 |
| **50%** | 2015.00 | 450000.00 | 60000.00 | 5.00 |
| **75%** | 2017.00 | 675000.00 | 98000.00 | 5.00 |
| **max** | 2020.00 | 10000000.00 | 2360457.00 | 14.00 |

아웃라이어가 몇 개 보입니다. 예를 들어 selling\_price에서 max값이 유독 높습니다. km\_driven은 max와 min 모두 아웃라이어로 보여집니다. 선형 모델은 아웃라이어에 대한 처리가 필요하지만 여기서는 트리 모델을 사용하므로 아웃라이어를 별도로 처리하지 않겠습니다.

## 9.3 전처리 : 텍스트 데이터

문자형 데이터를 숫자형으로 바꾸어야 연산이 쉽습니다. 그래서 첫 번째 작업으로는 단위 일치 및 숫자형으로 변환을 진행합니다. 단위가 섞여 있으므로 단위를 통일하고 그에 맞게 값을 계산을 해주고 단위에 해당하는 텍스트를 제거해 숫자형으로 변경하는 과정입니다. 두 번째는 텍스트 분류입니다. 불필요하게 구체적인 내용의 텍스트는 버리고, 필요한 부분만 남기겠습니다.

우선 숫자형 데이터로 변경할 컬럼들을 다루겠습니다. mileage, engine, max\_power, torque 변수에서 숫자와 문자가 혼재되어 있습니다. 숫자와 문자를 분리시키고 숫자의 자료형을 (소수점까지 표현해야 하므로) float형으로 지정해야 합니다.

문자형 데이터를 분리하는 데 판다스 시리즈에서 제공하는 str.split()을 사용합니다. 판다스 데이터프레임이 아닌 시리즈에만 있는 함수이기 때문에, 컬럼 하나씩만 인덱싱해 처리해줍니다.

<함수/>

| split() | 문자형 데이터를 분리하는 함수입니다. 기본 포맷은 다음과 같습니다.   | data['engine'].str.split() # 공백 기준으로 문자 분할 | | --- |   0 [1248, CC]  1 [1498, CC]  2 [1497, CC]  3 [1396, CC]  4 [1298, CC]  ...  8123 [1197, CC]  8124 [1493, CC]  8125 [1248, CC]  8126 [1396, CC]  8127 [1396, CC]  Name: engine, Length: 8128, dtype: object  기본적으로 빈 칸(띄어쓰기) 기준으로 분리시킵니다. 콤마, 마침표 같은 특정 문자를 기준으로 분리할 수도 있습니다. 괄호 안에 해당 문자를 입력하면 됩니다. 아래는 숫자 8을 기준으로 분리하는 예시 코드입니다.   | data['engine'].str.split('8') # 8 기준으로 문자 분할 | | --- | |
| --- | --- | --- | --- |

</>

### 9.3.1 engine 변수 전처리하기

다음과 같은 순서로 engine 변수의 값을 숫자와 단위(문자)로 분할해보겠습니다.



split() 함수는 분리 결과를 리스트 형태로 묶습니다. 우리는 숫자와 문자를 각각 별개의 컬럼에 두어야 하므로 expand 매개변수를 사용하겠습니다.

| data['engine'].str.split(expand=True) # 공백기준으로 문자를 분할하여 별도의 변수로 출력 |
| --- |

|  | **0** | **1** |
| --- | --- | --- |
| **0** | 1248 | CC |
| **1** | 1498 | CC |
| **2** | 1497 | CC |
| **3** | 1396 | CC |
| **4** | 1298 | CC |
| **...** | ... | ... |
| **8123** | 1197 | CC |
| **8124** | 1493 | CC |
| **8125** | 1248 | CC |
| **8126** | 1396 | CC |
| **8127** | 1396 | CC |

8128 rows × 2 columns

임의의 컬럼명 0과 1이라는 이름으로 숫자와 문자가 분리되었습니다. 숫자형 변수는 기존 engine 변수에 덮어쓰고, 문자형 변수는 engine\_unit이라는 새로운 변수 이름에 할당하겠습니다.

| data[['engine','engine\_unit']]= data['engine'].str.split(expand=True) # 분할된 문자들을 새로운 변수들로 저장 |
| --- |

이제 engine 변수를 확인하면 숫자만 남아있습니다.

| data['engine'].head() # 엔진 변수 확인 |
| --- |

0 1248

1 1498

2 1497

3 1396

4 1298

Name: engine, dtype: object

그러나 결과물을 보면 맨 아래에 dtype: object로 되어 있습니다. 즉, 숫자만 남았지만 이 데이터는 아직도 문자형 데이터로 인식되는 겁니다. astype() 함수를 이용하여 이를 숫자형 데이터로 변환해주겠습니다.

| data['engine'] = data['engine'].astype('float32') # 숫자형 변수로 변환 |
| --- |

그리고 다시 한번 head() 함수로 engine 컬럼을 확인해봅시다.

| data['engine'].head() # 엔진 변수 확인 |
| --- |

0 1248.0

1 1498.0

2 1497.0

3 1396.0

4 1298.0

Name: engine, dtype: float32

dtype이 float32로 변경되었습니다. 다음은 engine\_unit을 확인해봅시다. 문자 부분을 버리지 않고 굳이 남겨둔 이유는, 간혹 다른 단위를 가지는 데이터가 있을 수 있기 때문입니다. 우리가 head()로 확인한 engine의 데이터값에는 cc라는 단위만 있었습니다. 혹시 다른 확인하지 못한 다른 라인에 다른 단위가 붙어 있나 확인하고 넘어가겠습니다.

| data['engine\_unit'].unique() # 고윳값 확인 |
| --- |

array(['CC', nan], dtype=object)

CC라는 값만 한 가지만 존재합니다. 이제 해당 컬럼을 제거하겠습니다.

| data.drop('engine\_unit', axis=1, inplace= True) # 변수 제거 |
| --- |

### 9.3.2 max\_power 변수 전처리하기

engine 변수와 같은 방식으로 max\_power 변수도 변환하겠습니다.

| data[['max\_power','max\_power\_unit']] = data['max\_power'].str.split(expand=True) # 분할된 문자들을 새로운 변수들로 저장 |
| --- |

그리고 head()를 사용하여 확인해봅시다.

| data['max\_power'].head() # max\_power 변수 확인 |
| --- |

0 74

1 103.52

2 78

3 90

4 88.2

Name: max\_power, dtype: object

언뜻 보기에 잘 변환된 것으로 보입니다. 그러나 숫자형 변수로 변환하려고 아래 코드를 실행하면 에러가 발생합니다.

| data['max\_power'] = data['max\_power'].astype('float32') # 숫자형 변수로 변환 |
| --- |

**---------------------------------------------------------------------------**

**ValueError** Traceback (most recent call last)

**<ipython-input-587-cbd818a9f6ce>** in <module>

**----> 1** data**['max\_power']** **=** data**['max\_power'].**astype**('float32')**

**~\anaconda3\lib\site-packages\pandas\core\generic.py** in astype**(self, dtype, copy, errors)**

5875 **else:**

5876 **# else, only a single dtype is given**

**-> 5877** new\_data **=** self**.**\_mgr**.**astype**(**dtype**=**dtype**,** copy**=**copy**,** errors**=**errors**)**

5878 **return** self**.**\_constructor**(**new\_data**).**\_\_finalize\_\_**(**self**,** method**="astype")**

5879

**~\anaconda3\lib\site-packages\pandas\core\internals\managers.py** in astype**(self, dtype, copy, errors)**

629 self**,** dtype**,** copy**:** bool **=** **False,** errors**:** str **=** **"raise"**

630 ) -> "BlockManager":

**--> 631 return** self**.**apply**("astype",** dtype**=**dtype**,** copy**=**copy**,** errors**=**errors**)**

632

633 def convert(

**~\anaconda3\lib\site-packages\pandas\core\internals\managers.py** in apply**(self, f, align\_keys, ignore\_failures, \*\*kwargs)**

425 applied **=** b**.**apply**(**f**,** **\*\***kwargs**)**

426 **else:**

**--> 427** applied **=** getattr**(**b**,** f**)(\*\***kwargs**)**

428 **except** **(**TypeError**,** NotImplementedError**):**

429 **if** **not** ignore\_failures**:**

**~\anaconda3\lib\site-packages\pandas\core\internals\blocks.py** in astype**(self, dtype, copy, errors)**

671 vals1d **=** values**.**ravel**()**

672 **try:**

**--> 673** values **=** astype\_nansafe**(**vals1d**,** dtype**,** copy**=True)**

674 **except** **(**ValueError**,** TypeError**):**

675 **# e.g. astype\_nansafe can fail on object-dtype of strings**

**~\anaconda3\lib\site-packages\pandas\core\dtypes\cast.py** in astype\_nansafe**(arr, dtype, copy, skipna)**

1095 **if** copy **or** is\_object\_dtype**(**arr**)** **or** is\_object\_dtype**(**dtype**):**

1096 **# Explicit copy, or required since NumPy can't view from / to object.**

**-> 1097 return** arr**.**astype**(**dtype**,** copy**=True)**

1098

1099 **return** arr**.**view**(**dtype**)**

**ValueError**: could not convert string to float: 'bhp' ❶

이런 에러 메시지를 만나게 되면, 가장 먼저 확인할 부분은 ❶ 가장 아랫줄입니다. ValueError입니다. ‘bhp’라는 string을 float으로 바꿀 수 없다는 내용이군요. 진짜로 이런 데이터값이 있는지 보겠습니다.

| data[data['max\_power'] == 'bhp'] # max\_power 변수에 bhp라는 문자가 있는지 확인 |
| --- |

|  | **name** | **year** | **selling\_price** | **km\_driven** | **fuel** | **seller\_type** | **transmission** | **owner** | **mileage** | **engine** | **max\_power** | **torque** | **seats** | **max\_power\_unit** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **4933** | Maruti Omni CNG | 2000 | 80000 | 100000 | CNG | Individual | Manual | Second Owner | 10.9 km/kg | 796 | bhp | NaN | 8.0 | None |

4933번째 라인에 bhp가 들어 있습니다. 이는 원래 데이터에 숫자 없이 bhp만 있어서 str.split()에서 빈 칸을 기준으로 분리할 때 분리되지 않은 채로 남은 겁니다. 지금은 단 한 건의 데이터이기 때문에 간단히 라인을 제거하거나 해당 값을 Null로 변환할 수 있습니다. 하지만 잘못된 데이터가 여러 건 있을 때를 대비하여 Try and Except 블록을 사용해 처리하겠습니다.

<글상자/>

Try and Except 블록은 다음과 같이 별도의 함수 안에 넣어 활용할 수 있습니다.

| def isFloat(value): # 함수 정의  try: # ❶ 시도  num = float(value) # 값을 숫자로 변환  return num # 변환된 값 리턴  except ValueError: # ❷ try에서 ValueError가 난 경우  return np.NaN # np.NaN 리턴 |
| --- |

Try and Except 구문의 기본 개념은 if절과 비슷합니다. ❶ try 블록 코드를 실행하고 ➝ ❷ 에러가 발생하면 except 블록 코드를 실행합니다. 여기에서 try 블록에는 간단하게 해당 값들을 숫자로 변환할 수 있도록 float() 함수를 사용했습니다. 에러가 발생했던 ‘bhp’는 숫자로 변환할 수 없으므로 except 블록으로 넘어갑니다. except 블록에는 발생되는 에러의 종류를 지정해야 하고, 지정된 에러가 발생했을 때만 블록 안의 코드가 실행됩니다. 우리는 위의 에러 메시지에서 이 에러가 ValueError에 해당하는 것을 확인했으므로 except ValueError:라고 지정하고 Null값을 반환하도록 np.NaN을 지정했습니다. 이제 isFloat() 함수를 다음과 같이 적용시키면 에러 없이 숫자 형태로 변환할 수 있습니다.

| data['max\_power'] = data['max\_power'].apply(isFloat) # isFloat 함수를 사용하여 숫자형 변수로 변환 |
| --- |

max\_power\_unit도 확인하겠습니다.

| data['max\_power\_unit'].unique() # 고윳값 확인 |
| --- |

array(['bhp', nan, None], dtype=object)

‘bhp’ 이외에 Null값들이 있습니다. 적어도 다른 단위는 아니므로 따로 조치해줄 필요는 없습니다. 이 컬럼도 제거하겠습니다.

| data.drop('max\_power\_unit', axis=1, inplace=True) # 변수 제거 |
| --- |

### 9.3.3 mileage 변수 전처리하기

mileage 변수를 같은 방법으로 숫자와 문자로 분리하겠습니다.

| data[['mileage','mileage\_unit']] = data['mileage'].str.split(expand=True) # 분할된 문자들을 새로운 변수들로 저장 |
| --- |

숫자만 남은 mileage 변수를 숫자형 데이터로 변환합니다.

| data['mileage'] = data['mileage'].astype('float32') # 숫자형 변수로 변환 |
| --- |

단위를 확인하겠습니다.

| data['mileage\_unit'].unique() # 고윳값 확인 |
| --- |

array(['kmpl', 'km/kg', nan], dtype=object)

‘kmpl’과 ‘km/kg’ 단위가 있군요. 단위가 다른 숫자를 동일한 컬럼의 값으로 활용하는 것은 적절하지 않기 때문에, 단위를 이해하고 숫자를 변환해야 합니다. kmpl은 km/l, 즉 리터당 킬로미터를 의미하고, km/kg는 킬로그램당 킬로미터를 뜻합니다. 여기서 킬로미터는 주행거리이고, 리터와 킬로그램은 연료에 대한 측정 단위입니다. 휘발유나 디젤은 리터 단위로 측정하지만 LGP나 CNG는 킬로그램 단위이기 때문에 이런 두 가지 단위가 나타난 겁니다. 두 단위의 근본적인 차이는 연료의 종류 때문입니다. fuel는 연료를 명시한 컬럼입니다. fuel 컬럼을 활용해 해결해봅시다. 같은 kmpl이라고 휘발유인지 디젤인지도 구분해야 하기 때문에 fuel을 이용하는 방법이 더 적절합니다.

fuel에 어떤 종류의 연료들이 있는지 확인하겠습니다.

| data['fuel'].unique() # 고윳값 확인 |
| --- |

array(['Diesel', 'Petrol', 'LPG', 'CNG'], dtype=object)

연료 종류는 4가지입니다. 다른 종류의 연료로 주행거리를 비교하려면 같은 기준을 세워야 합니다. 연료의 가격을 활용하는 방법은 어떤가요? 즉 1달러당 몇 km를 주행할 수 있는지를 본다면 연료의 종류와 상관없이 동등한 비교가 될 겁니다.

구글에서 각 연료에 대한 동일 시점의 가격을 검색하여 다음과 같은 정보를 얻었습니다.

* Petrol : 리터당 $80.43
* Diesel : 리터당 $73.56
* LPG : 킬로그램당 $40.85
* CNG : 킬로그램당 $44.23

mileage 변수를 각 연료별 가격으로 나누면 1달러당 주행거리가 됩니다. 다음과 같은 함수를 만들어주겠습니다.

| def mile(x): # 함수 정의  if x['fuel'] == 'Petrol': # fuel이 Petrol이면  return x['mileage'] / 80.43 # mileage를 80.43으로 나눔  elif x['fuel'] == 'Diesel': # fuel이 Diesel이면  return x['mileage'] / 73.56 # mileage를 73.56으로 나눔  elif x['fuel'] == 'LPG': # fuel이 LPG이면  return x['mileage'] / 40.85 # mileage를 40.85로 나눔  else: # 그 밖에는  return x['mileage'] / 44.23 # mileage를 44.23으로 나눔 |
| --- |

이 함수에서는 fuel과 mileage 두 변수를 사용합니다. 그래서 함수를 호출할 때 입력값 x는 data 전체이어야 합니다.

data에 함수를 적용해줍니다.

| data['mileage'] = data.apply(mile, axis=1) # mile 함수로 마일리지 수정 |
| --- |

apply()를 이용하여 함수를 적용할 때, 데이터 전체가 입력되는 상황에서는 axis 매개변수를 1로 지정해야 합니다. 그렇지 않으면 함수 내의 변수명을 컬럼에서 찾지 않고 인덱스에서 찾습니다(drop() 함수를 사용할 때 axis 매개변수를 사용하는 것과 같은 원리입니다).

이제 milage\_unit 컬럼은 필요가 없으므로 제거합니다.

| data.drop('mileage\_unit',axis=1,inplace=True) # 변수 제거 |
| --- |

### 9.3.4 torque 변수 전처리하기

마지막으로 torque 변수 분할해줍시다. 과정은 다음과 같이 2단계입니다. ❶ 단위 앞부분 숫자만 추출해 숫자형으로 바꾼 후, ❷ Nm 단위로 스케일링합니다.



head()를 사용하여 다시 한번 어떤 데이터가 들어있는지 확인해봅시다.

| data['torque'].head() # torque 변수 5줄 확인 |
| --- |

0 190Nm@ 2000rpm

1 250Nm@ 1500-2500rpm

2 12.7@ 2,700(kgm@ rpm)

3 22.4 kgm at 1750-2750rpm

4 11.5@ 4,500(kgm@ rpm)

Name: torque, dtype: object

@ 단위는 공통적으로 rpm이 있고, 그 앞에 Nm 혹은 kgm이 있습니다. rpm의 경우 어떤 데이터는 특정값(예 : 2000rpm)이고, 어떤 데이터는 범위(예 : 1500-2500rpm)로 들어 있기 때문에 여기서는 rpm은 제외하고 nm과 kgm 부분만 다루어보겠습니다.

이 변수는 수많은 숫자와 문자가 다양한 규칙으로 배열되어 있기 때문에 이전 방법으로는 숫자와 문자를 구분할 수 없습니다. 여러 방법이 있겠으나, string을 다루는 방법으로 처리해보겠습니다.

우선 대소문자가 섞인 형태이므로 str.upper() 함수로 모두 대문자로 변환해주겠습니다(소문자로 바꾸시려면 upper 대신 lower()를 쓰세요).

| data['torque'] = data['torque'].str.upper() # torque 변수 대문자로 변환 |
| --- |

그리고 단위를 따로 빼내는 작업을 하겠습니다. 이를 위해 해당 값 안에 NM 혹은 KGM이 있는지를 판별하여 해당 단위를 반환하는 함수를 만들겠습니다.

<글상자>

**in으로 문자열에서 특정 문자가 있는지 판단하는 방법**

| 'ex' in 'example' # example에 ex가 있는지 확인 |
| --- |

True

| 'abc' in 'example' # example에 abc가 있는지 확인 |
| --- |

False

‘ex’는 ‘example’ 안에 포함되어 있으므로 True, ‘abc’는 ‘example’ 안에 포함되지 않으므로 False가 출력됩니다.

</>

in을 사용해 함수를 만들어봅시다.

| def torque\_unit(x): # 함수 정의  if 'NM' in str(x): # x에 NM이 있으면  return 'Nm' # Nm을 리턴  elif 'KGM' in str(x): # x에 KGM이 있으면  return 'kgm' # kgm을 리턴 |
| --- |

위 함수를 torque 변수에 적용시키고, 결과물을 torque\_unit이라는 이름의 새로운 컬럼으로 저장하겠습니다.

| data['torque\_unit'] = data['torque'].apply(torque\_unit) # torque변수를 torque\_unit 함수에 적용하여 torque\_unit이라는 변수로 저장 |
| --- |

torque\_unit에 단위값이 잘 들어갔는지 확인해봅시다.

| data['torque\_unit'].unique() # torque\_unit 고윳값 확인 |
| --- |

array(['Nm', 'kgm', None], dtype=object)

Nm과 kgm이 잘 들어있고, Null값(None)들도 존재합니다. 데이터 자체가 아예 없어서 Null일 수도 있지만, Nm이나 kgm에 해당하지 않는 단위여서 Null일 수도 있으니 확인해볼 필요가 있습니다. 우선 isna()로 torque\_unit 컬럼이 Null인 부분을 필터링하여, torque 변수의 unique()를 체크하겠습니다. 세분화하여 단계별로 설명하자면

data['torque\_unit'].isna() # ❶

data[data['torque\_unit'].isna()] # ❷

data[data['torque\_unit'].isna()]['torque'].unique() # ❸

❶ torque\_unit이 Null값인지 확인합니다. ❷ 얻은 결과를 data[ ]안에 넣어 필터링합니다.

❸ 필터링된 데이터에서 torque의 unique값을 확인합니다.

즉, 다음과 같이 한 줄로도 쓸 수도 있습니다.

| # torque\_unit이 결측치인 라인의 torque 변수 고윳값 확인  data[data['torque\_unit'].isna()]['torque'].unique() |
| --- |

array(['NAN', '250@ 1250-5000RPM', '510@ 1600-2400', '110(11.2)@ 4800',

'210 / 1900'], dtype=object)

결과물을 보니 Nm도 kgm도 없는 데이터와 NAN으로 입력된 데이터값이 있습니다. 우선 Nm도 kgm도 써있지 않은 데이터가 Nm이나 kgm 중 하나에 속한다는 전제하에 추론해봅시다. 다른 데이터를 보면 Nm는 보통 100, 200 등 백 단위이고, kgm은 10, 20 등 십 단위 숫자입니다. 숫자 크기를 고려하면 모두 Nm에 해당한다고 추론할 수 있습니다. 그럼 torque\_unit의 빈 값을 모두 Nm으로 채우겠습니다.

| data['torque\_unit'].fillna('Nm', inplace= True) # 결측치를 Nm으로 대체 |
| --- |

이제 torque 변수에서 맨 앞 숫자 부분을 빼내야 합니다. for문을 사용하여 각 string의 맨 앞자리부터 하나씩 체크하여, 숫자가 아닌 부분이 나왔을 때 해당 부분의 위치를 가지고 인덱싱으로 잘라내겠습니다. 여기서는 숫자뿐만 아니라 소수점도 고려하여 마침표까지 확인해야 합니다.

예를 들어 다음과 같은 문자열이 있다고 합니다.

| string\_example = '12.7@ 2,700(KGM@ RPM)' # 예제 스트링 정의 |
| --- |

문자열을 살펴보면 점과 숫자가 아닌 곳은 4번째 텍스트입니다.

* 0번째 텍스트 : ‘1’
* 1번째 텍스트 : ‘2’
* 2번째 텍스트 : ‘.’
* 3번째 텍스트 : ‘7’
* 4번째 텍스트 : ‘@’

4번째 위치에서 숫자도 마침표도 아닌 @이 나옵니다. 4번째 위치이므로 4라는 숫자로 인덱싱을 해 자르면 원하는 숫자(12.7)가 나올 것 같습니다.

| string\_example[:4] # 4번째 자리 전까지 인덱싱 |
| --- |

'12.7'

원하는 결과가 나왔습니다. 이렇게 숫자만 빼오는 함수를 만들어봅시다. 이것을 for문으로 쉽게 구현하려면 enumerate() 함수를 함께 써야 합니다.

enumerate()는 for문에서 하나의 데이터를 불러올 때마다 몇 번째 호출값인지 숫자를 함께 제공합니다. 그래서 기존 반환값을 받는 변수가 두 개(i, j)입니다. i는 0부터 시작하여 for문이 끝날 때까지 반복되는 숫자만큼 올라가며, j는 해당 위치의 값을 보여줍니다.

| for i, j in enumerate(string\_example):  print(i,'번째 텍스트: ', j) |
| --- |

0 번째 텍스트: 1

1 번째 텍스트: 2

2 번째 텍스트: .

3 번째 텍스트: 7

4 번째 텍스트: @

5 번째 텍스트:

6 번째 텍스트: 2

7 번째 텍스트: ,

8 번째 텍스트: 7

9 번째 텍스트: 0

10 번째 텍스트: 0

11 번째 텍스트: (

12 번째 텍스트: K

13 번째 텍스트: G

14 번째 텍스트: M

15 번째 텍스트: @

16 번째 텍스트:

17 번째 텍스트: R

18 번째 텍스트: P

19 번째 텍스트: M

20 번째 텍스트: )

enumerate() 함수를 사용해 각 값이 숫자 혹은 마침표에 속하는지 판별해 숫자 혹은 마침표가 아닌 것 중 가장 첫 번째로 나타나는 자리를 찾아내겠습니다.

| for i,j in enumerate(string\_example): # 인덱스(순서)를 포함한 순회  if j not in '0123456789.': # ❶ 만약 j가 0123456789.에 포함되지 않으면  cut = i # 인덱스(순서)를 cut에 저장  break # 순회 중지 |
| --- |

❶ j로 불러온 문자가 ‘0123456789.’에 **없는지** not in을 사용해 확인합니다. ‘0123456789.’ 중 하나에 해당하지 않으면 해당 순번인 숫자 i를 cut에 저장한 뒤 break를 만나 for문을 종료합니다(break를 사용하지 않으면 for문이 계속 돌아가면서 cut을 새로운 i값으로 업데이트 해버리기 때문에 숫자도 마침표도 아닌 것 중 가장 마지막 자리의 인덱스를 가지게 됩니다). 위의 코드를 실행한 결과 cut에는 4번째 텍스트를 지칭하는 4가 저장됩니다.

위 코드를 함수로 만들겠습니다.

| def split\_num(x): # 함수 정의  x = str(x) # ❶ 문자 형태로 변환  for i,j in enumerate(x): # 인덱스를 포함한 순회  if j not in '0123456789.': # j가 0123456789.에 속하지 않으면  cut = i # 인덱스를 cut에 저장  break # 순회 중지  return x[:cut] # ❷ cut 이전 자리까지 인덱싱하여 리턴 |
| --- |

❶ 입력값 x가 확실히 string이 되도록 x = str(x)로 지정합니다. 함수를 적용하기 위해 apply()를 썼을 때, 만약 특정값이 숫자로만 구성되어 있다면 함수 내에서 string이 아닌 float이나 int형으로 인식하게 되고, for문에서 에러가 발생하기 때문에 미리 조치를 해둔 겁니다.

❷ 반환값은 cut을 사용하여 주어진 스트링을 인덱싱해야 합니다. x[:cut]도 x가 string이 아니라 float 혹은 int 타입이라면 인덱싱이 불가능하므로, 위에 언급한 string으로의 변환 과정이 꼭 필요합니다.

위 함수를 torque 변수에 적용시키고, 결괏값을 torque변수에 업데이트하겠습니다.

| data['torque'] = data['torque'].apply(split\_num) # torque 변수를 split\_num 함수에 적용 |
| --- |

head() 함수로 데이터가 잘 변환되었는지 확인해봅시다.

| data['torque'] # torque 변수 확인 |
| --- |

0 190

1 250

2 12.7

3 22.4

4 11.5

...

8123 113.7

8124 24

8125 190

8126 140

8127 140

Name: torque, Length: 8128, dtype: object

원하는 대로 잘 변환되었습니다. 이제 단위를 맞추어 변환하기 전에, 해당 값을 float형으로 변환해줍니다.

| data['torque'] = data['torque'].astype('float64') # 숫자 형태로 변환 |
| --- |

**---------------------------------------------------------------------------**

**ValueError** Traceback (most recent call last)

**<ipython-input-45-99a6d6768f59>** in <module>

**----> 1** data**['torque']** **=** data**['torque'].**astype**('float64')**

**~\anaconda3\lib\site-packages\pandas\core\generic.py** in astype**(self, dtype, copy, errors)**

5875 **else:**

5876 **# else, only a single dtype is given**

**-> 5877** new\_data **=** self**.**\_mgr**.**astype**(**dtype**=**dtype**,** copy**=**copy**,** errors**=**errors**)**

5878 **return** self**.**\_constructor**(**new\_data**).**\_\_finalize\_\_**(**self**,** method**="astype")**

5879

**... 중략 ...**

**~\anaconda3\lib\site-packages\pandas\core\dtypes\cast.py** in astype\_nansafe**(arr, dtype, copy, skipna)**

1095 **if** copy **or** is\_object\_dtype**(**arr**)** **or** is\_object\_dtype**(**dtype**):**

1096 **# Explicit copy, or required since NumPy can't view from / to object.**

**-> 1097 return** arr**.**astype**(**dtype**,** copy**=True)**

1098

1099 **return** arr**.**view**(**dtype**)**

**ValueError**: could not convert string to float: ''

이번에도 아까와 비슷한 에러가 발생했습니다. string을 float으로 변경할 수 없다는 것인데, 비어 있는 string값 ''이 있는 것 같습니다. 앞서처럼 try and except 블록을 이용할 수도 있지만, 여기서는 간단히 Null로 대체한 뒤 진행하겠습니다. Numpy를 이용한 np.NaN으로 지정해주면 됩니다.

| data['torque'] = data['torque'].replace('', np.NaN) # ''를 결측치로 대체 |
| --- |

그럼 float으로 변환하는 코드를 다시 실행하겠습니다.

| data['torque'] = data['torque'].astype('float64') # 숫자 형태로 변환 |
| --- |

에러 없이 수행이 완료되었으면 head()를 실행해 확인하겠습니다.

| data['torque'].head() # torque 변수 5줄 확인 |
| --- |

0 190.0

1 250.0

2 12.7

3 22.4

4 11.5

Name: torque, dtype: float64

이제 단위에 따른 차이를 맞춰주는 변환을 해야 합니다. Nm과 kgm은 중력과 관련된 것으로 kgmｘ9.8066 = Nm입니다. 이를 활용하여 kgm일 때는 9.80660을 곱한 값을 반환하고 kgm이 아니면, 즉 Nm이면 그대로 반환하여 모든 값을 Nm으로 변환하는 함수를 만들겠습니다. 이번에도 함수 내에 두 변수가 사용되기 때문에, 입력은 데이터프레임 전체를 받는 x로 전제하고 만들겠습니다.

| def torque\_trans(x): # 함수 정의  if x['torque\_unit'] == 'kgm': # torque\_unit이 kgm이면  return x['torque'] \* 9.8066 # torque 변수에 9.8066 곱하여 리턴  else: # 아닐 경우  return x['torque'] # torque 변수 그냥 리턴 |
| --- |

데이터프레임 전체가 입력이기 때문에 apply()로 적용할 때 이번에도 axis 매개변수를 꼭 넣어줍니다.

| # torque 변수에 torque\_trans 함수 적용  data['torque'] = data.apply(torque\_trans, axis=1) |
| --- |

변환이 완료되었으면 torque\_unit은 더는 필요하지 않으므로 삭제하겠습니다.

| data.drop('torque\_unit', axis=1, inplace=True) # 변수 제거 |
| --- |

이제 문자와 숫자가 혼재된 변수들의 정리가 끝났습니다. data.head()를 통해 변환된 모습을 살펴보겠습니다.

| data.head() # 상위 5줄 확인 |
| --- |

|  | **name** | **year** | **selling\_price** | **km\_driven** | **fuel** | **seller\_type** | **transmission** | **owner** | **mileage** | **engine** | **max\_power** | **torque** | **seats** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Maruti Swift Dzire VDI | 2014 | 450000 | 145500 | Diesel | Individual | Manual | First Owner | 0.318108 | 1248 | 74 | 190 | 5.0 |
| **1** | Skoda Rapid 1.5 TDI Ambition | 2014 | 370000 | 120000 | Diesel | Individual | Manual | Second Owner | 0.287384 | 1498 | 103.52 | 250 | 5.0 |
| **2** | Honda City 2017-2020 EXi | 2006 | 158000 | 140000 | Petrol | Individual | Manual | Third Owner | 0.220067 | 1497 | 78 | 124.54382 | 5.0 |
| **3** | Hyundai i20 Sportz Diesel | 2010 | 225000 | 127000 | Diesel | Individual | Manual | First Owner | 0.31267 | 1396 | 90 | 219.66784 | 5.0 |
| **4** | Maruti Swift VXI BSIII | 2007 | 130000 | 120000 | Petrol | Individual | Manual | First Owner | 0.200174 | 1298 | 88.2 | 112.7759 | 5.0 |

### 9.3.5 name 변수 전처리하기

이번에는 name 변수를 다뤄보겠습니다. 자동차의 브랜드/모델명이 기입된 변수입니다. 수많은 자동차 모델을 다 담아내면 좋겠지만, 그렇게 되면 더미 변수를 만들었을 때 수많은 컬럼들이 추가되므로 좋지 않습니다. 꼭 필요하다면 수많은 컬럼이 생기더라도 유지하는 것이 좋습니다만, 어떤 모델 이름이냐보다는 그 모델이 가지고 있는 자동차의 특성, 즉 engine, max\_power, torque 등이 더 중요하다고 볼 수 있습니다. 다만 여기에서 한 가지 가지고 가야 할 것은 브랜드입니다. 동일 스펙의 자동차라도 비싼 브랜드의 경우 가격이 더 비쌀 수밖에 없으므로, 모델명을 버리더라도 브랜드명은 꼭 가져가는 것이 좋겠습니다. 그럼 name 변수에서 브랜드명만 따로 분리하겠습니다. head()에서 보이는 값들을 보면 브랜드 명은 맨 앞에 위치해 있고 띄어쓰기로 모델명과 구분되어 있으므로 str.split()으로 쉽게 분리할 수 있습니다. 다만, 모델명에 수많은 띄어쓰기들이 있으므로 이를 모두 컬럼으로 만들면 불필요한 수많은 컬럼이 나타납니다. 불필요한 컬럼은 따로 저장할 필요가 없으므로 가장 첫 컬럼인 [0]만 인덱싱하여 브랜드 이름을 name 변수에 그대로 덮어쓰기 해주겠습니다.

| data['name'] = data['name'].str.split(expand=True)[0] # name변수를 공백으로 나누고 첫번째 부분으로 업데이트 |
| --- |

혹시 브랜드 이름이 아닌 다른 값들이 잘못 분리되어 있을지 모르니, unique()를 호출해 브랜드 이름만 들어있는지 확인해봅시다.

| data['name'].unique() # 고윳값 확인 |
| --- |

array(['Maruti', 'Skoda', 'Honda', 'Hyundai', 'Toyota', 'Ford', 'Renault',

'Mahindra', 'Tata', 'Chevrolet', 'Fiat', 'Datsun', 'Jeep',

'Mercedes-Benz', 'Mitsubishi', 'Audi', 'Volkswagen', 'BMW',

'Nissan', 'Lexus', 'Jaguar', 'Land', 'MG', 'Volvo', 'Daewoo',

'Kia', 'Force', 'Ambassador', 'Ashok', 'Isuzu', 'Opel', 'Peugeot'],

dtype=object)

띄어쓰기로 인하여 Land Rover가 Land로만 저장된 것 이외에 모두 제대로 된 브랜드 이름입니다. Land라는 이름으로 저장됐다고 해도 모델링에는 영향이 없으나, 연습삼아 해당 이름을 Land Rover로 변경해주고 넘어가겠습니다. replace() 함수를 이용하면 간단히 변경할 수 있으며, 첫째 매개변수는 변경 대상, 두 번째 매개변수는 변경하려는 값을 넣어주면 됩니다.

| data['name'] = data['name'].replace('Land', 'Land Rover') # Land를 Land Rover로 대체 |
| --- |

나머지 텍스트 데이터는 더미 변수로 변환하면 되는데, 이는 결측치를 처리한 후에 진행하겠습니다.

## 9.4 전처리 : 결측치 처리와 더미 변수 변환

어떤 변수에 얼마만큼의 결측치가 있는지 살펴봅시다.

| data.isna().mean() # 변수별 결측치의 평균 확인 |
| --- |

name 0.000000

year 0.000000

selling\_price 0.000000

km\_driven 0.000000

fuel 0.000000

seller\_type 0.000000

transmission 0.000000

owner 0.000000

mileage 0.027190

engine 0.027190

max\_power 0.026575

torque 0.027313

seats 0.027190

dtype: float64

mileage부터 그 아래로 모두 결측치가 있고, 약 2~3% 정도의 수준입니다. mileage, engine, seats의 경우 결측치 비율이 동일한 것으로 보아 특정 행에서 mileage 이하 컬럼들이 전혀 수집되지 않은 것으로 보입니다. 모두 자동차의 스펙과 관련된 변수들인데, 수집되지 않았다면 예측하는 데 큰 문제가 있을 수 있습니다. 스펙 관련된 모든 값을 평균값으로 치환할 경우 오히려 노이즈 역할 만할 가능성도 높습니다. 결측치의 비율 또한 약 2% 수준으로 높지 않기 때문에, 여기서는 과감하게 해당 행을 모두 제거하겠습니다.

| data.dropna(inplace= True) # 결측치 행 제거 len(data) # 데이터 길이 확인 |
| --- |

7906

만약 제거에 확신이 서지 않는 상황이라면, 제거하는 방법과 평균치로 치환하는 방법 모두를 사용해보시고 모델링 결과를 비교해보아도 좋습니다.

이제 결측치가 제거되었으므로 남은 텍스트 컬럼을 더미 변수로 변환하겠습니다. 대상이 되는 변수는 name, fuel, seller\_type, transmission, owner입니다.

| data = pd.get\_dummies(data, columns = ['name','fuel','seller\_type','transmission','owner'], drop\_first=True) # 더미 변수로 변환 |
| --- |

변환이 완료되었으면 head()를 통해 최종 데이터셋을 확인해보시기 바랍니다.

| data.head() # 상위 5줄 확인 |
| --- |

|  | **year** | **selling\_price** | **km\_driven** | **mileage** | **engine** | **max\_power** | **torque** | **seats** | **name\_Ashok** | **name\_Audi** | **...** | **fuel\_Diesel** | **fuel\_LPG** | **fuel\_Petrol** | **seller\_type\_Individual** | **seller\_type\_Trustmark Dealer** | **transmission\_Manual** | **owner\_Fourth & Above Owner** | **owner\_Second Owner** | **owner\_Test Drive Car** | **owner\_Third Owner** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2014 | 450000 | 145500 | 0.318108 | 1248 | 74 | 190 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **1** | 2014 | 370000 | 120000 | 0.287384 | 1498 | 103.52 | 250 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| **2** | 2006 | 158000 | 140000 | 0.220067 | 1497 | 78 | 124.54382 | 5 | 0 | 0 | ... | 0 | 0.0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| **3** | 2010 | 225000 | 127000 | 0.312670 | 1396 | 90 | 219.66784 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **4** | 2007 | 130000 | 120000 | 0.200174 | 1298 | 88.2 | 112.7759 | 5 | 0 | 0 | ... | 0 | 0.0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

## 9.5 모델링 및 평가하기

우선 훈련셋과 시험셋을 나누어주도록 합시다.

| from sklearn.model\_selection import train\_test\_split # 임포트 X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.drop('selling\_price', axis=1), data['selling\_price'], test\_size=0.2, random\_state=100) # 훈련셋/시험셋 분리 |
| --- |

랜덤 포레스트 모델은 사이킷런에서 ensemble 패키지 안에 속해 있습니다. 결정 트리와 마찬가지로 RandomForestRegressor와 RandomForestClassifier를 제공하는데, 이번 데이터는 연속형 변수가 타깃이므로 RandomForestRegressor를 사용하겠습니다.

| from sklearn.ensemble import RandomForestRegressor # 임포트 |
| --- |

모델링 및 예측 방법은 기존 다른 모델과 같습니다.

| model = RandomForestRegressor(random\_state=100) # ❶ 모델 객체 생성 model.fit(X\_train, y\_train) # ❷ 학습 train\_pred = model.predict(X\_train) # 훈련셋 예측 test\_pred = model.predict(X\_test) # 시험셋 예측 |
| --- |

❶ 랜덤 포레스트는 매번 다른 방식으로 나무들을 생성하므로 random\_state를 지정하는 것이 좋습니다. ❷ 훈련셋으로 학습한 후, ❸ 훈련셋과 시험셋에 대한 예측값 두 가지를 모두 구해봅시다.

종속 변수가 연속형 데이터이니, 실제값과 예측값의 차이를 합산하는 RMSE를 사용해 평가하겠습니다.

| from sklearn.metrics import mean\_squared\_error # 임포트 print("train\_rmse:", mean\_squared\_error(y\_train, train\_pred) \*\* 0.5, "test\_rmse:", mean\_squared\_error(y\_test, test\_pred) \*\* 0.5) # rmse 확인 |
| --- |

train\_rmse: 53531.41548125947 test\_rmse: 131855.18391308116

## 9.6 이해하기 : K-폴드 교차검증

교차검증cross validation의 목적은 모델의 예측력을 더 안정적으로 평가하기 위함입니다(교차타당성이라고도 합니다). 새로운 데이터를 얼마나 잘 예측하는지 확인하고자 훈련셋과 시험셋을 나누어서 평가했고, 이러한 데이터 분할은 랜덤 샘플링으로 이루어졌기 때문에 어느 정도 안정적인 보장을 받습니다. 하지만 랜덤 샘플링으로 나누어졌더라도, 그에 따라 발생하는 우연에 의한 오차들이 예측력을 평가하는 데 작은 노이즈로 존재하는 것도 사실입니다. 여러분이 train\_test\_split()에서 random\_state 매개변수에 10개의 다양한 숫자를 넣어보면 매번 훈련셋의 평가값(RMSE, accuracy score 등)이 계속 변하는 것을 볼 수 있을 겁니다. 이러한 작은 오차들까지 고려하여 평가하는 방법이 바로 교차검증입니다.

<용어/>

**교차검증**

다양한 훈련셋/시험셋을 통하여 모델에 더 신뢰할 수 있는 평가를 하는 방법

</>

K-폴드 교차검증K-fold Cross-Validation의 아이디어는 간단합니다. 데이터를 특정 개수(K개)로 쪼개어서 그중 하나씩을 선택하여 시험셋으로 사용하되, 이 과정을 K번 만큼 반복하는 겁니다. 예를 들어 데이터를 5개로 나누었다면, 아래 예시의 그림과 같습니다.

<note/>

K-폴드 교차검증를 K-겹 교차검증이라고도 불러요.

</>



훈련셋 시험셋

이터레이션 1 오차1

이터레이션 2 오차2

이터레이션 3 오차3 오차 =

이터레이션 4 오차4

이터레이션 5 오차5

각 이터레이션(반복)은 모델링을 수행하는 단위입니다. 이터레이션 1에서는 맨 마지막 데이터 조각(흰색)이 시험셋으로 사용되고 나머지(파란색)가 훈련셋으로 사용되어 모델링 및 평가가 이뤄집니다. 여기서 얻어진 평가값이 오차1입니다. 다음 이터레이션 2에서는 네 번째 데이터 조각이 시험셋, 나머지가 훈련셋이며, 여기서 모델링 및 평가가 이뤄진 결과는 오차2입니다. 5개의 데이터로 분류했으므로 총 5개의 시험셋을 만들어 모델링 및 평가 과정을 5번 반복할 수 있습니다. 이렇게 얻어진 평가값(오차) 총 5개를 평균 내어 최종 평가값(RMSE)을 도출합니다.

이 과정을 직접 코딩할 수도 있지만, 이미 사이킷런에서 패키지(KFold)로 제공합니다. KFold 패키지는 사이킷런 라이브러리의 model\_selection에서 제공합니다.

| from sklearn.model\_selection import KFold # 임포트 |
| --- |

KFold를 사용하기에 앞서, 우리가 가진 데이터의 인덱스를 한 번 정리해줄 필요가 있습니다. data를 호출하여 현재 데이터의 인덱스를 살펴봅시다.

| data # data 확인 |
| --- |

|  | **year** | **selling\_price** | **km\_driven** | **mileage** | **engine** | **max\_power** | **torque** | **seats** | **name\_Ashok** | **name\_Audi** | **...** | **fuel\_Diesel** | **fuel\_LPG** | **fuel\_Petrol** | **seller\_type\_Individual** | **seller\_type\_Trustmark Dealer** | **transmission\_Manual** | **owner\_Fourth & Above Owner** | **owner\_Second Owner** | **owner\_Test Drive Car** | **owner\_Third Owner** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2014 | 450000 | 145500 | 0.318108 | 1248 | 74 | 190 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **1** | 2014 | 370000 | 120000 | 0.287384 | 1498 | 103.52 | 250 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| **2** | 2006 | 158000 | 140000 | 0.220067 | 1497 | 78 | 124.54382 | 5 | 0 | 0 | ... | 0 | 0.0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| **3** | 2010 | 225000 | 127000 | 0.312670 | 1396 | 90 | 219.66784 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **4** | 2007 | 130000 | 120000 | 0.200174 | 1298 | 88.2 | 112.7759 | 5 | 0 | 0 | ... | 0 | 0.0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **8123** | 2013 | 320000 | 110000 | 0.230014 | 1197 | 82.85 | 113.7 | 5 | 0 | 0 | ... | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **8124** | 2007 | 135000 | 119000 | 0.228385 | 1493 | 110 | 235.3584 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| **8125** | 2009 | 382000 | 120000 | 0.262371 | 1248 | 73.9 | 190 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **8126** | 2013 | 290000 | 25000 | 0.320419 | 1396 | 70 | 140 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **8127** | 2013 | 290000 | 25000 | 0.320419 | 1396 | 70 | 140 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

7906 rows × 48 columns

❶ 왼쪽 인덱스를 보면 0부터 8127까지 있지만, ❷ 실제로는 7906줄입니다. 이는 우리가 중간에 dropna()를 사용하여 약 2% 정도 되는 줄을 제거했기 때문입니다. KFold는 이 인덱스 값을 이용하여 데이터를 분할하는데, 이와 같이 중간에 빈 값이 존재하면 에러가 발생하므로, reset\_index()를 사용하여 인덱스를 정리해주겠습니다.

| data.reset\_index(drop=True, inplace=True) # 인덱스를 변수로 가져옴 |
| --- |

❶ drop 매개변수를 사용하지 않으면 기존 인덱스가 새로운 컬럼 형태로 추가되기 때문에 drop=True를 꼭 입력해줍니다.

다시 data를 호출하겠습니다.

| data # data 확인 |
| --- |

|  | **year** | **selling\_price** | **km\_driven** | **mileage** | **engine** | **max\_power** | **torque** | **seats** | **name\_Ashok** | **name\_Audi** | **...** | **fuel\_Diesel** | **fuel\_LPG** | **fuel\_Petrol** | **seller\_type\_Individual** | **seller\_type\_Trustmark Dealer** | **transmission\_Manual** | **owner\_Fourth & Above Owner** | **owner\_Second Owner** | **owner\_Test Drive Car** | **owner\_Third Owner** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2014 | 450000 | 145500 | 0.318108 | 1248 | 74 | 190 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **1** | 2014 | 370000 | 120000 | 0.287384 | 1498 | 103.52 | 250 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| **2** | 2006 | 158000 | 140000 | 0.220067 | 1497 | 78 | 124.54382 | 5 | 0 | 0 | ... | 0 | 0.0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| **3** | 2010 | 225000 | 127000 | 0.312670 | 1396 | 90 | 219.66784 | 5 | 0 | 0 | ... | 1 | 0.0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **4** | 2007 | 130000 | 120000 | 0.200174 | 1298 | 88.2 | 112.7759 | 5 | 0 | 0 | ... | 0 | 0.0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **7901** | 2013 | 320000 | 110000 | 0.230014 | 1197 | 82.85 | 113.7 | 5 | 0 | 0 | ... | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **7902** | 2007 | 135000 | 119000 | 0.228385 | 1493 | 110 | 235.3584 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| **7903** | 2009 | 382000 | 120000 | 0.262371 | 1248 | 73.9 | 190 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **7904** | 2013 | 290000 | 25000 | 0.320419 | 1396 | 70 | 140 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **7905** | 2013 | 290000 | 25000 | 0.320419 | 1396 | 70 | 140 | 5 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

7906 rows × 48 columns

이제 ❶ 마지막 줄의 인덱스가 7905로 빈 값이 없게 배열되었습니다. 다음은 KFold()에서 몇 개의 데이터로 분할할지 지정해주고 그 속성을 kf라는 이름으로 저장합니다. 여기서는 5개로 분할해보겠습니다.

| kf = KFold(n\_splits=5) # KFold 객체 생성 |
| --- |

그리고 독립변수와 종속변수를 각각 X와 y에 할당해줍니다.

| X = data.drop('selling\_price', axis=1) # 종속 변수 제거하여 X에 저장 y = data['selling\_price'] # 종속 변수를 y에 저장 |
| --- |

kf에 내재되어 있는 split() 함수로 데이터인 X를 분할하고, 이 결괏값을 출력하겠습니다.

| for i, j in kf.split(X): # 순회  print(i, j) # i,j 출력 |
| --- |

[1582 1583 1584 ... 7903 7904 7905] [ 0 1 2 ... 1579 1580 1581]

[ 0 1 2 ... 7903 7904 7905] [1582 1583 1584 ... 3160 3161 3162]

[ 0 1 2 ... 7903 7904 7905] [3163 3164 3165 ... 4741 4742 4743]

[ 0 1 2 ... 7903 7904 7905] [4744 4745 4746 ... 6322 6323 6324]

[ 0 1 2 ... 6322 6323 6324] [6325 6326 6327 ... 7903 7904 7905]

kf에는 이미 5개로 분할한다는 정보가 저장되어 있기 때문에, for문을 사용하면 총 5개의 다른 인덱스를 보여줍니다. 또한 분할을 하면 훈련셋과 시험셋으로 사용할 인덱스가 따로 나오기 때문에 i, j 두 값으로 받아줘야 합니다.

첫 번째 줄은 이터레이션 1에 사용될 인덱스이고 순서대로 이터레이션 5에 사용될 인덱스까지 준비되어 있습니다. 그리고 왼쪽 리스트는 훈련셋에 사용될 인덱스이며, 오른쪽의 리스트는 시험셋에 사용될 인덱스입니다.

이 값들을 준비한 X와 y에 loc를 사용하여 인덱싱해주겠습니다. 아래 코드에서는 직관적인 이해를 위하여 i, j 대신 train\_index, test\_index라는 이름으로 대체했습니다. 참고로 이 코드는 아직 완성된 코드가 아닌 일부이며, 실행해도 아무런 결괏값이 보여지지 않습니다.

| for train\_index , test\_index in kf.split(X): # 순회  X\_train , X\_test = X.loc[train\_index],X.loc[test\_index] # ❶❷ X\_trian, X\_test 설정  y\_train , y\_test = y[train\_index] , y[test\_index] # ❸ y\_train, y\_test 설정 |
| --- |

❶ X를 loc로 인덱싱해 X\_train에 저장합니다. 인덱싱 숫자는 train\_index에서 가져왔습니다.

❷ X를 loc로 인덱싱해 X\_test에 저장됩니다. 인덱싱 숫자는 test\_index에서 가져옵니다.

❸ y는 시리즈 형태이므로 loc를 사용하지 않고 곧바로 인덱싱할 수 있습니다. y\_train과 y\_test에 인덱싱한 결과를 저장합니다.

이렇게 X\_train, X\_test, y\_train, y\_test 값들을 각각 저장할 수 있으며, for문이 5번 반복되기 때문에 매번 다른 훈련셋과 시험셋이 설정됩니다. 이제 위의 코드에 덧붙여서 모델링과 평가를 위한 코드까지 추가하겠습니다.

이에 앞서, 최종적으로 train rmse와 test rmse 값들이 총 5개씩 결과물로 얻어지므로 이를 리스트 형태로 저장할 빈 리스트를 먼저 만들겠습니다.

| train\_rmse\_total = [] # train rmse를 위한 빈 리스트 생성 test\_rmse\_total = [] # test rmse를 위한 빈 리스트 생성 |
| --- |

그리고 for문 안에 훈련과 평가를 위한 코드도 추가합니다. 가장 마지막 부분에는 빈 리스트에 train rmse와 test rmse를 각각 추가할 수 있도록 append()를 사용했습니다.

| for train\_index , test\_index in kf.split(X): # 순회  X\_train , X\_test = X.loc[train\_index],X.loc[test\_index] # X\_train, X\_test 정의  y\_train , y\_test = y[train\_index] , y[test\_index] # y\_train, y\_test 정의  model = RandomForestRegressor(random\_state=100) # 모델 객체 생성  model.fit(X\_train,y\_train) # 학습  train\_pred = model.predict(X\_train) # 훈련셋 예측  test\_pred = model.predict(X\_test) # 시험셋 예측    train\_rmse = mean\_squared\_error(y\_train, train\_pred) \*\* 0.5 # 훈련셋 rmse 계산  test\_rmse = mean\_squared\_error(y\_test, test\_pred) \*\* 0.5 # 시험셋 rmse 계산    train\_rmse\_total.append(train\_rmse) # 리스트에 훈련셋 rmse 추가  test\_rmse\_total.append(test\_rmse) # 리스트에 시험셋 rmse 추가 |
| --- |

위 코드가 정상 실행됐다면 train\_rmse\_total에서 총 5개의 RMSE를 볼 수 있습니다.

| train\_rmse\_total # 리스트에 모아진 rmse 확인 |
| --- |

[50825.5556350298,

58854.04054344074,

57904.19615940739,

56218.23740006373,

58967.150857632456]

최종 RMSE는 이것들의 평균이므로 평균값을 구해야 합니다. 리스트에서는 mean() 함수가 내재되어 있지 않으므로 다음과 같이 합과 나눗셈으로 평균값을 구해줍니다. 학습 RMSE(train rmse)와 평가 RMSE(test rmse)를 함께 출력하겠습니다.

print("train\_rmse:", sum(train\_rmse\_total)/5, "test\_rmse:",sum(test\_rmse\_total)/5)

train\_rmse: 56553.836119114814 test\_rmse: 142936.58918244042

교차검증을 사용하기 전과 약간의 차이가 보입니다. 전반적으로 RMSE가 더 올라간 모습인데, 앞서 교차검증을 사용한 결과가 조금 더 정확한 평가 결과입니다.

## 9.7 이해하기 : 랜덤 포레스트

랜덤 포레스트는 결정 트리의 집합체입니다. 결정 트리를 사용하면 오버피팅 문제가 쉽게 발생하는데, 랜덤 포레스트는 여러 트리를 활용하여 최종 모델을 만듭니다. 즉 다양한 트리의 의견을 반영하기 때문에 오버피팅 위험을 낮출 수 있습니다. 랜덤 포레스트가 여러 개의 트리를 만들 때는 데이터 전체를 사용하지 않고, 매번 다른 일부의 데이터를 사용하여 다른 트리를 만들어냅니다. 당연히 일부 데이터만 사용하면 전체 데이터를 사용한 결과보다 예측력이 떨어지겠지만, 예측력이 다소 떨어지는 수많은 트리들이 함께 모여 중간값을 찾아내면 오버피팅을 막는 데 효율적입니다.

랜덤 포레스트가 일부 데이터를 취하는 기준은 크게 두 가지입니다.

첫 번째는 데이터의 행row 기준으로 일부씩만 취하여 사용합니다. 예를 들어 1만 행의 데이터가 있다고 할 때, 1만 행 전체를 사용하는 게 아닌 약 2/3에 해당하는 데이터만을 사용합니다. 각 트리에서는 매번 다르게 추출된 2/3에 해당하는 데이터가 사용되어, 다른 결과의 트리가 만들어지는 겁니다.

두 번째는 열column, 즉 변수 기준입니다. 각 트리에서는 주어진 모든 독립변수들을 사용하지 않고, 그중 일부의 변수들만을 매번 다르게 추출하여 각 트리를 만들어냅니다. 이런 방식이 가지는 장점은 특정 변수의 강력한 힘을 어느 정도 통제할 수 있다는 겁니다. 예를 들어 A라는 변수가 예측하는 데 매우 결정적인 역할을 한다면 트리 1000개를 만들더라도 최상위 노드에서의 분류 기준은 항상 A 변수가 될 겁니다. 즉 다른 트리 1000개를 만들어도 큰 차이가 없는 트리들이 나오는 겁니다. 하나의 변수에 너무 큰 무게가 실린 모델은 역시 오버피팅을 불러올 수 있으므로, 몇몇 트리에서 해당 변수를 제외해서 다른 변수들에도 더 무게가 실릴 수 있도록 해주면 오버피팅을 피하는 데 도움이 됩니다.



랜덤 포레스트의 최종 예측값은 각 트리의 예측값들을 기반으로 만들어집니다. 회귀 문제는 연속형 변수를 예측하기 때문에, 각 트리에서 만들어낸 예측값들의 평균값을 랜덤 포레스트의 최종 예측값으로 사용합니다. 분류 문제는 각 트리에서 예측한 값들 중 최다 투표값으로 랜덤 포레스트의 예측값이 결정됩니다. 예를 들어 0과 1을 분류하는 문제에서 총 1000개 트리 중 700개 트리가 1로 예측을 하고 300개 트리가 0으로 예측했다면, 가장 많은 투표를 받은 1이 랜덤 포레스트의 최종 예측값이 됩니다.

## 9.8 하이퍼파라미터 튜닝

랜덤 포레스트는 수많은 하이퍼파라미터를 가지고 있습니다. 여기서는 주로 사용될 주요 매개변수만 설명을 한 뒤 이를 활용하여 다시 모델링하겠습니다.

* n\_estimators : 랜덤 포레스트를 구성하는 결정 트리의 개수입니다. 기본값은 100으로 설정되어 있습니다. 너무 많거나 적은 수를 입력하면 성능이 떨어지므로 적정 수준의 값을 찾아서 넣어야 합니다.
* max\_depth : 결정 트리와 동일하게, 각 트리의 최대 깊이를 제한합니다. 숫자가 낮을수록 오버피팅을 피할 수 있으며, 또한 언더피팅의 위험도 올라갑니다.
* min\_samples\_split : 해당 노드를 나눌 것인지 말 것인지를 노드 데이터 수를 기준으로 판단합니다. 이 매개변수에 지정된 숫자보다 적은 수의 데이터가 노드에 있으면 더는 분류하지 않습니다. 숫자가 높을수록 분리되는 노드가 적어질 것이므로, 오버피팅을 피하는 방법이자 언더피팅의 위험도 있습니다. 기본값은 2입니다.
* min\_samples\_leaf : 분리된 노드의 데이터에 최소 몇 개의 데이터가 있어야 할지를 결정하는 매개변수입니다. 여기에 지정된 숫자보다 적은 수의 데이터가 분류된다면, 해당 분리는 이루어지지 않습니다. 마찬가지로 숫자가 클수록 오버피팅을 피할 수 있고, 언더피팅의 위험도는 높아집니다. 기본값은 1입니다.
* n\_jobs : 병렬 처리에 사용되는 CPU 코어 수입니다. 많은 코어를 사용할수록 속도가 빨라지며, -1을 입력하면 지원하는 모든 코어를 사용합니다. 기본값은 None으로, 실제는 1개 코어를 사용합니다. 랜덤 포레스트의 속도가 다소 느린만큼 충분한 코어를 사용하는 게 좋습니다.

max\_depth,min\_sample\_split, min\_sample\_leaf는 모두 비슷한 특성을 지닌 매개변수로 상호 연관되어 있습니다. 따라서 숫자를 적절하게 분배하지 않으면 의미없는 매개변수로 남을 수 있습니다. 예를 들어 max\_depth를 너무 작게 가져가고 min\_sample\_split에도 작은 수를 넣었다면, min\_sample\_split에 해당하는 수의 노드가 등장하기 전에 트리의 분류가 끝나버려서 min\_sample\_split이 아무런 역할도 하지 못할 수도 있습니다.

아래는 하이퍼파라미터 튜닝의 예시입니다. n\_estimators에 300, max\_depth에 50, min\_sample\_split에 5, min\_sample\_leaf에 1(기본값)을 넣어주었습니다.

| train\_rmse\_total = [] # 훈련셋 rmse를 위한 빈 리스트 생성 test\_rmse\_total = [] # 시험셋 rmse를 위한 빈 리스트 생성  for train\_index , test\_index in kf.split(X): # 순회  X\_train , X\_test = X.iloc[train\_index,:],X.iloc[test\_index,:] # X\_train, X\_test 정의  y\_train , y\_test = y[train\_index] , y[test\_index] # y\_train, y\_test 정의  model = RandomForestRegressor(n\_estimators = 300, max\_depth = 50, min\_samples\_split = 5, min\_samples\_leaf = 1, n\_jobs=-1, random\_state=100) # 하이퍼파라미터를 지정하여 모델 객체 생성  model.fit(X\_train,y\_train) # 학습  train\_pred = model.predict(X\_train) # 훈련셋 예측  test\_pred = model.predict(X\_test) # 시험셋 예측    train\_rmse = mean\_squared\_error(y\_train, train\_pred) \*\* 0.5 # 훈련셋 rmse 계산  test\_rmse = mean\_squared\_error(y\_test, test\_pred) \*\* 0.5 # 시험셋 rmse 계산    train\_rmse\_total.append(train\_rmse) # 훈련셋 rmse를 리스트에 추가  test\_rmse\_total.append(test\_rmse) # 시험셋 rmse를 리스트에 추가 |
| --- |

train rmse와 test rmse를 확인하겠습니다.

| print("train\_rmse:", sum(train\_rmse\_total)/5, "test\_rmse:",sum(test\_rmse\_total)/5) # 리스트에 모인 rmse의 평균값 계산 |
| --- |

train\_rmse: 66762.84568886801 test\_rmse: 142205.83441414658

train rmse는 하이퍼파라미터 튜닝 전보다 다소 높아졌으나, test rmse는 조금 더 낮아졌습니다. 조금이나마 오버피팅이 줄어들으니 새로운 데이터를 예측하기에는 더 좋은 모델이라 할 수 있습니다.

하이퍼파라미터에 가장 적합한 숫자들의 조합을 찾는 것은 쉬운 일이 아닙니다. 여기에서는 임의의 숫자를 넣은 예시로 설명을 마치지만, 다음 장에서는 더 효율적으로 다양한 조합을 테스트해볼 수 있는 그리드 서치를 소개하겠습니다.

## 학습 마무리

#### 되짚어보기

9.1 중고차의 가격을 예측하는 모델을 만들어봅니다.

9.2 판다스, 넘파이, 맷플롯립, 시본 라이브러리를 임포트했습니다. 프로젝트에 쓸 예제 데이터셋을 불러옵니다.

9.3 수치로 활용할 수 있는 변수가 문자로 되어 있습니다. 숫자 단위를 분리했습니다.

9.4 결측치가 있는 행들을 제거하고, 카테고리형 변수들을 더미 변수로 변환했습니다.

9.5 랜덤 포레스트를 사용하여 예측 모델을 만들었습니다. 연속형 변수이기 때문에 아직 RMSE를 단적으로 ‘좋다/나쁘다’ 평가할 순 없습니다.

9.6 훈련셋과 시험셋의 다양한 조합을 통하여 더 신뢰할 만한 결과를 얻은 K겹 교차검증을 학습했습니다

9.7 랜덤 포레스트는 결정 트리의 진화된 버전으로, 여러 독립된 결정 트리들이 합심하여 예측을 해냅니다.

9.8 랜덤 포레스트에서 쓸 수 있는 몇가지 하이퍼파라미터를 변경해보았습니다. 그 결과 test rmse(평균 제곱근 편차)를 조금 낮출 수 있었습니다.



#### 과제

자동차 데이터셋을 랜덤 포레스트으로 학습해 자동차 가격을 예측해보았습니다. 하이퍼파라미터를 다양하게 변경하여 더 낮은 rmse를 구할 수 있는지 시도해보세요. 결정 트리를 이용하여 모델링해보고, 랜덤 포레스트의 결과와 비교해봅시다.

#### 핵심 용어 정리

1. **교차검증** : 다양한 훈련셋/시험셋을 통하여 모델에 더 신뢰할 수 있는 평가를 하는 방법
2. **앙상블 기법** : 여러 모델을 만들고 각 예측값들을 투표/평균 등으로 통합하여 더 정확한 예측을 도모하는 방법

#### 새로운 함수와 라이브러리

* **split()** : 문자형 데이터를 분리하는 함수입니다. 기본값은 공백 기준으로 분리합니다.
* **Try and Except 구문** : 함수에 에러가 발생했을 경우 대응할 방법을 정의할 수 있습니다.
* **KFold()** : 교차검증 알고리즘 객체 생성

## 연습문제

1. str.split() 함수에서, 결과를 각각의 새로운 변수로 나타내는 매개변수는?

① columns

② expand

③ index

④ n\_splits

2. 다음 K-폴드 교차검증에 대한 설명 중 옳지 않은 것은?

① 데이터셋을 여러 개로 분할하여, 다양한 평가셋/시험셋으로 모델을 학습하고 평가한다.

② 매개변수 n\_splits을 10으로 지정하면 10개의 다른 평가셋/시험셋을 만들 수 있다.

③ K-폴드 교차검증을 사용하면, 사용하지 않을 때보다 항상 더 낮은 RMSE를 얻을 수 있다.

④ K-폴드 교차검증을 통해 얻은 결과가 더 신뢰할 만한 결과라고 볼 수 있다.

3. 다음 랜덤 포레스트에 대한 설명 중 옳지 않은 것은?

① 트리 기반의 알고리즘이다.

② 여러 트리를 활용하여 평균/다수결로 최종 예측 결과를 얻는다.

③ 학습 속도가 결정 트리보다 빠르다.

④ 결정 트리에 비해 모델의 해석이 어렵다.

#### 정답 및 해설

1. 2

② expand

2. 3

③ K-폴드 교차검증을 사용하면, 사용하지 않을 때보다 항상 더 낮은 RMSE를 얻을 수 있다. ← K-폴드 교차검증을 사용하면 경우에 따라 더 높은 RMSE가 나올 수도 있습니다. K-폴드 교차검증은 더 낮은 RMSE를 얻기 위함이 아니라, 우연의 요소가 배제된 더 신뢰할 만한 결과를 얻기 위함입니다.

3. 3

③ 학습 속도가 결정 트리보다 빠르다. ← 랜덤 포레스트는 여러 트리를 생성하기 때문에 결정 트리에 비해 학습 속도가 느립니다.