Машинное обучение

Дополнительные материалы

Материалы > Анализ и обработка данных

Продолжим работу в том же блокноте

Pipeline и ColumnTransformer

ColumnTransformer

Ранее мы рассмотрели применение пайплайна для последовательного преобразования данных и обучения модели. При этом, обратите внимание, у нас были только количественные данные (собственно, один количественный признак LSTAT).

Что делать, если у нас есть как количественные, так и категориальные признаки, и им соответственно нужны разные преобразования (более того, разным количественным и категориальным признакам также могут понадобиться разные преобразования)?

Здесь выручает **ColumnTransformer**. Он позволяет «прописать» отдельным признакам (т.е. столбцам, columns) свои преобразования, а затем объединить результат и передать в модель. Рассмотрим пример.

```
# создадим датасет с данными о клиентах банка
scoring = {
    'Name' : ['Иван', 'Николай', 'Алексей', 'Александра', 'Евгений', 'Елена'],
    'Age' : [35, 43, 21, 34, 24, 27],
    'Experience' : [7, 13, 2, np.nan, 4, 12],
    'Salary' : [95, 135, 73, 100, 78, 110],
    'Credit_score' : ['Good', 'Good', 'Bad', 'Medium', 'Medium', 'Good'],
    'Outcome' : [1, 1, 0, 1, 0, 1]
}

scoring = pd.DataFrame(scoring)
scoring
```

	Name	Age	Experience	Salary	Credit_score	Outcome
0	Иван	35	7.0	95	Good	1
1	Николай	43	13.0	135	Good	1
2	Алексей	21	2.0	73	Bad	0
3	Александра	34	NaN	100	Medium	1
4	Евгений	24	4.0	78	Medium	0
5	Елена	27	12.0	110	Good	1

```
# разобьем данные на признаки и целевую переменную
X = scoring.iloc[:, 1 :-1]
y = scoring.Outcome

# поместим название количественных и категориальных признаков в списки
num_col = ['Age', 'Experience', 'Salary']
cat_col = ['Credit_score']

# ColumnTransformer позволяет применять разные преобразователи к разным столбцам
from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer

# создадим объекты преобразователей для количественных
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy = 'mean')

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

# и категориального признака
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder(categories = [['Bad', 'Medium', 'Good']])

# поместим их в отдельные пайплайны
num_transformer = make_pipeline(imputer, scaler)
cat_transformer = make_pipeline(encoder)

# поместим пайплайны в ColumnTransformer
preprocessor = ColumnTransformer(
      transformers=[('num', num_transformer, num_col),
                    ('cat', cat_transformer, cat_col)])

# создадим объект модели, которая будет использовать все признаки
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()

# создадим еще один пайплайн, который будет включать объект ColumnTransformer и
# объект модели
pipe = make_pipeline(preprocessor, model)

pipe.fit(X, y)

# сделаем прогноз
pipe.predict(X)
```

```
array([1, 1, 0, 1, 0, 1])
```

Библиотека joblib

Сохранение пайплайна

Библиотека joblib позволяет сохранить пайплайн в файл примерно так, как мы поступали с моделями и библиотекой pickle.

```
import joblib

# сохраним пайплайн в файл с расширением .joblib
joblib.dump(pipe, 'pipe.joblib')

# импортируем из файла
new_pipe = joblib.load('pipe.joblib')
```

```
8
9  # обучим модель и сделаем прогноз
10 new_pipe.fit(X, y)
11 pipe.predict(X)
```

```
array([1, 1, 0, 1, 0, 1])
```

Кэширование функции

В качестве небольшого дополнения рассмотрим возможность joblib кэшировать, например, созданную нами функцию. Кэширование существенно ускоряет время ее исполнения.

```
import time

# напишем функцию, которая принимает список чисел
# и выдает их квадрат
def square_range(start_num, end_num):

  res = []
  # пройдемся по заданному перечню
  for i in range(start_num, end_num):
    res.append(i ** 2)
    # искусственно замедлим исполнение
    time.sleep(0.5)

  return res

start = time.time()
res = square_range(1, 21)
end = time.time()

# посмотрим на время исполнения и финальный результат
print(end - start)
print(res)
```

```
10.014686584472656
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400
```

Поместим функцию в кэш и вызовем из кэша в первый раз.

```
# определим, куда мы хотим сохранить кэш
location = '/content/'

# используем класс Memory
memory = joblib.Memory(location, verbose = 0)

def square_range_cached(start_num, end_num):

  res = []
  # пройдемся по заданному перечню
  for i in range(start_num, end_num):
    res.append(i ** 2)
    # искусственно замедлим исполнение
    time.sleep(0.5)

  return res

# поместим в кэш
square_range_cached = memory.cache(square_range_cached)

start = time.time()
res = square_range_cached(1, 21)
end = time.time()
```

```
24
25 print(end - start)
26 print(res)
```

```
1 10.015617370605469
2 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400
```

Вызовем из кэша еще раз.

```
1 start = time.time()
2 res = square_range_cached(1, 21)
3 end = time.time()
4
5 print(end - start)
6 print(res)
```

```
1 0.0011799335479736328
2 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400
```

Параллелизация

Параллелизация (parallelization) или параллельное исполнение кода на нескольких процессорах (CPU) может существенно увеличить скорость выполнения операций. Посмотрим, сколько процессоров доступно в Google Colab.

```
1 n_cpu = joblib.cpu_count()
2 n_cpu
```

```
1 2
```

Напишем медленную функцию.

```
1 def slow_square(x):
2   time.sleep(1)
3   return x ** 2
```

Применим эту функцию к числам от 0 до 9.

```
1 %time [slow_square(i) for i in range(10)]
```

```
1 CPU times: user 54.5 ms, sys: 7.15 ms, total: 61.7 ms
2 Wall time: 10 s
3 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Выполним параллелизацию.

```
1  from joblib import Parallel, delayed
2
3  # функция delayed() разделяет исполнение кода на несколько задач (функций)
4  delayed_funcs = [delayed(slow_square)(i) for i in range(10)]
5
6  # класс Parallel отвечает за параллелизацию
7  # если указать n_jobs = -1, будут использованы все доступные CPU
8  parallel_pool = Parallel(n_jobs = n_cpu)
9
10 %time parallel_pool(delayed_funcs)
```

```
1 CPU times: user 56.4 ms, sys: 47.8 ms, total: 104 ms
2 Wall time: 6.01 s
3 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
1 # для наглядности выведем задачи, созданные функцией delayed()
2 delayed_funcs
```

```
[(<function __main__.slow_square(x)>, (0,), {}),
 (<function __main__.slow_square(x)>, (1,), {}),
 (<function __main__.slow_square(x)>, (2,), {}),
 (<function __main__.slow_square(x)>, (3,), {}),
 (<function __main__.slow_square(x)>, (4,), {}),
 (<function __main__.slow_square(x)>, (5,), {}),
 (<function __main__.slow_square(x)>, (6,), {}),
 (<function __main__.slow_square(x)>, (7,), {}),
 (<function __main__.slow_square(x)>, (8,), {}),
 (<function __main__.slow_square(x)>, (9,), {})]
```

Встраивание функций и классов в sklearn

Библиотека sklearn предоставляет инструменты для создания собственных функций и классов, которые затем можно встроить в классы sklearn в соответствии с их парадигмой (т.е. с методами **.fit()**, **.transform()** и т.д.).

Встраивание функций

Напишем собственную функцию, которая будет кодировать категориальную переменную в соответствии с переданным ей словарем.

```
# напишем простой encoder
# будем передавать в функцию данные, столбец, который нужно кодировать,
# и схему кодирования (map)
def encoder(df, col, map_dict):
  df_map = df.copy()
  df_map[col] = df_map[col].map(map_dict)
  return df_map
```

```
# зададим схему кодирования столбца Credit_score
map_dict = {'Bad' : 0,
            'Medium' : 1,
            'Good': 2}
```

Теперь импортируем класс FunctionTransformer, которому при создании соответствующего объекта передадим созданную функцию и ее параметры в виде словаря (параметр kw_args).

```
from sklearn.preprocessing import FunctionTransformer

encoder = FunctionTransformer(func = encoder,
                              kw_args = {'col' : 'Credit_score',
                                         'map_dict' : map_dict})
```

FunctionTransformer автоматически создаст стандартные методы класса sklearn, в частности, метод **.fit_transform()**.

```
encoder.fit_transform(X)
```

	Age	Experience	Salary	Credit_score
0	35	7.0	95	2
1	43	13.0	135	2
2	21	2.0	73	0
3	34	NaN	100	1
4	24	4.0	78	1

5	27	12.0	110	2

Встраивание классов

Помимо функций в парадигму sklearn можно встраивать и классы. Для этого новый класс должен наследовать классы BaseEstimator и TransformerMixin. Создадим такой же encoder, но на этот раз в виде класса.

```
# класс BaseEstimator создает методы .get_params() и .set_params()
# класс TransformerMixin создает .fit_transform()
from sklearn.base import BaseEstimator, TransformerMixin

class Encode(BaseEstimator, TransformerMixin):

  # при создании объекта класса передаем столбец и схему кодирования
  def __init__(self, col, map_dict):
    self.col = col
    self.map_dict = map_dict

  def fit(self, df, y = None):
    return self

  def transform(self, df, y = None):
    df_map = df.copy()
    # применим преобразование
    df_map[self.col] = df_map[self.col].map(self.map_dict)
    return df_map

  def inverse_transform(self, df, y = None):
    df_inv_map = df.copy()
    # поменяем ключи и значения местами
    inv_map = {v: k for k, v in self.map_dict.items()}
    # применим обратное преобразование
    df_inv_map[self.col] = df_inv_map[self.col].map(inv_map)
    return df_inv_map
```

Создадим объект класса Encode и применим метод **.fit_transform()**, который мы не объявляли явным образом.

```
encoder = Encode('Credit_score', dict(Bad = 0, Medium = 1, Good = 2))

X_trans = encoder.fit_transform(X)
X_trans
```

	Age	Experience	Salary	Credit_score
0	35	7.0	95	2
1	43	13.0	135	2
2	21	2.0	73	0
3	34	NaN	100	1
4	24	4.0	78	1
5	27	12.0	110	2

Попробуем метод **.inverse_transform()**.

```
encoder.inverse_transform(X_trans)
```

	Age	Experience	Salary	Credit_score
0	35	7.0	95	Good
1	43	13.0	135	Good
2	21	2.0	73	Bad
3	34	NaN	100	Medium
4	24	4.0	78	Medium
5	27	12.0	110	Good

Такие вновь созданные классы sklearn, в частности, можно встраивать в pipeline.

```
imputer = SimpleImputer(strategy = 'mean')
scaler = StandardScaler()

encoder = Encode('Credit_score', dict(Bad = 0, Medium = 1, Good = 2))

num_transformer = make_pipeline(imputer, scaler)
cat_transformer = make_pipeline(encoder)

preprocessor = ColumnTransformer(
      transformers=[('num', num_transformer, num_col),
                    ('cat', cat_transformer, cat_col)])

model = LogisticRegression()

pipe = make_pipeline(preprocessor, model)
pipe.fit(X, y)
pipe.predict(X)
```

```
array([1, 1, 0, 1, 0, 1])
```

Подведем итог

В дополнительных материалах мы изучили инструмент ColumnTransformer, некоторые возможности библиотеки joblib, а также встраивание собственных функций и классов в парадигму sklearn.