**فصل 5: مدیریت داده‌های رسته‌ای[[1]](#footnote-1)**

**5. مقدمه**

اغلب مفید است که اشیا را نه از نظر کمیت، بلکه از نظر کیفیت اندازه‌گیری کنیم. ما اغلب این اطلاعات کیفی را به عنوان عضویت یک نمونه در یک دسته مجزا مانند جنسیت، رنگ‌ها یا برند خودرو نشان می‌دهیم. با این حال، همه داده‌های رسته‌ای یکسان نیستند. مجموعه‌ای از رسته‌ها بدون ترتیب ذاتی، اسمی[[2]](#footnote-2) نامیده می‌شوند. نمونه‌هایی از رسته‌های اسمی عبارتند از:

* آبی، قرمز، سبز
* مرد، زن
* موز، توت فرنگی، سیب

در مقابل، وقتی مجموعه‌ای از مقوله‌ها دارای ترتیب طبیعی هستند، از آن به عنوان ترتیبی[[3]](#footnote-3) یاد می‌کنیم. مثلا:

* کم، متوسط، ​​زیاد
* جوان، پیر
* موافق، خنثی، مخالف

علاوه بر این، اطلاعات دسته‌بندی اغلب در داده‌ها به‌عنوان بردار یا ستونی از رشته‌ها نشان داده می‌شوند (مانند «Maine»، «Texas»، «Delaware»). مشکل این است که بیشتر الگوریتم‌های یادگیری ماشین به ورودی‌هایی نیاز دارند که مقادیر عددی باشند. الگوریتم k نزدیکترین همسایه یک مثال ساده ارائه می دهد. یک مرحله در الگوریتم محاسبه فواصل بین مشاهدات است، اغلب با استفاده از فاصله اقلیدسی:

که در آن x و y دو مشاهدات هستند و زیرمجموعه i مقدار ویژگی مشاهدات را نشان می دهد. با این حال، محاسبه فاصله آشکارا غیرممکن است اگر مقدار x یک رشته باشد (به عنوان مثال، " Texas"). در عوض، باید رشته را به فرمت عددی تبدیل کنیم تا بتوان آن را در معادله فاصله اقلیدسی وارد کرد. هدف ما ایجاد تحولی است که به درستی اطلاعات موجود در دسته ها را منتقل کند (تعارف، فواصل نسبی بین دسته ها و غیره). دراین فصل، تکنیک‌هایی را برای ایجاد این تحول و همچنین غلبه بر چالش‌هایی که اغلب هنگام مدیریت داده‌های طبقه‌بندی با آن مواجه می‌شوند، پوشش خواهیم داد.

**1.5رمزگذاری ویژگی­های دسته بندی اسمی**

**مسئله**

شما یک ویژگی با کلاس­های اسمی دارید که ترتیب ذاتی ندارد (به عنوان مثال، سیب، گلابی، موز).

**راه حل**

با استفاده ازکتابخانه scikit-learn’s ، ویژگی را یک‌بار OHE با LabelBinarizer می­کنیم.

*# Import libraries*

import numpy as np

from sklearn.preprocessing import LabelBinarizer,

MultiLabelBinarizer

*# Create feature*

feature = np.array( [ ["Texas"],

["California"],

["Texas"],

["Delaware"],

["Texas"] ] )

*# Create one-hot encoder*

one\_hot = LabelBinarizer()

*# One-hot encode feature*

one\_hot.fit\_transform(feature)

array ( [ [0, 0, 1],

[1, 0, 0],

[0, 0, 1],

[0, 1, 0],

[0, 0, 1] ] )

ما می توانیم از متد classes برای خروجی کلاس ها استفاده کنیم:

*# View feature classes*

one\_hot.classes\_

array( [ 'California', 'Delaware', 'Texas' ],

dtype =' < U10')

اگر بخواهیم رمزگذاری OHE را معکوس کنیم، می توانیم از inverse\_transform استفاده کنیم:

*# Reverse one-hot encoding*

one\_hot.inverse\_transform(one\_hot.transform(feature))

array( [ 'Texas', 'California', 'Texas', 'Delaware', 'Texas'], dtype =' <U10 ')

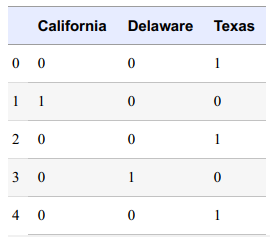
ما حتی می توانیم از pandas برای رمزگذاری یکباره این ویژگی استفاده کنیم.

*# Import library*

import pandas as pd

*# Create dummy variables from feature*

pd.get\_dummies(feature[:,0])



یکی از توانایی‌های مفید scikit-learn موقعیتی است که در آن هر یک از آنها چندین کلاس را فهرست می کند:

*# Create multiclass feature*

multiclass\_feature = [ ("Texas", "Florida"),

("California", "Alabama"),

("Texas", "Florida"),

("Delware", "Florida"),

("Texas", "Alabama") ]

*# Create multiclass one-hot encoder*

one\_hot\_multiclass = MultiLabelBinarizer()

*# One-hot encode multiclass feature*

one\_hot\_multiclass.fit\_transform(multiclass\_feature)

array ( [ [0, 0, 0, 1, 1],

[1, 1, 0, 0, 0],

[0, 0, 0, 1, 1],

[0, 0, 1, 1, 0],

[1, 0, 0, 0, 1] ] )

یک بار دیگر می‌توانیم کلاس‌ها را با متد classes ببینیم:

*# View classes*

one\_hot\_multiclass.classes\_

array([ 'Alabama', 'California', 'Delware', 'Florida', 'Texas'], dtype =object)

**بحث**

ممکن است فکر کنیم استراتژی مناسب این است که به هر کلاس یک مقدار عددی اختصاص دهیم (به عنوان مثال، Texas = 1, California = 2)

با این حال، وقتی کلاس‌های ما ترتیب ذاتی ندارند (به عنوان مثال، Texas «کم‌تر از California » نیست)، مقادیر عددی ما به اشتباه ترتیبی را ایجاد می‌کنند که وجود ندارد. استراتژی مناسب ایجاد یک ویژگی باینری برای هر کلاس در ویژگی اصلی است. این اغلب رمزگذاری یک‌طرفه (در ادبیات یادگیری ماشین) یا ساختگی (در ادبیات آماری و تحقیقاتی) نامیده می‌شود. ویژگی راه حل ما یک بردار حاوی سه کلاس (یعنی 'California', 'Florida', 'Texas') بود.

در رمزگذاری OHE، هر کلاس به ویژگی خاص خود تبدیل می‌شود که در صورت ظاهر شدن کلاس، 1 و در غیر این صورت 0 می‌شود. از آنجا که ویژگی ما دارای سه کلاس بود، OHE سه ویژگی باینری (یکی برای هر کلاس) را برگرداند. با استفاده از OHE ، می‌توانیم عضویت یک مشاهده را در یک کلاس ثبت کنیم و در عین حال این مفهوم را حفظ کنیم که کلاس فاقد هر نوع سلسله مراتبی است. در نهایت، شایان ذکر است که اغلب توصیه می‌شود که پس از رمزگذاری یکباره یک ویژگی، یکی از ویژگی‌های OHE را در ماتریس حاصل رها کنیم تا از وابستگی خطی جلوگیری کنیم.

همچنین

* [Dummy Variable Trap، Algosome](http://bit.ly/2FvVJkC)
* [رها کردن یکی از ستون ها هنگام استفاده از OHE](http://bit.ly/2FwrxG0)

**2.5 رمزگذاری دسته بندی ترتیبی**

**مسئله**

شما یک ویژگی دسته بندی ترتیبی دارید (به عنوان مثال، زیاد، متوسط، کم).

**راه حل**

با استفاده از راه حل از روش جایگزینی DataFrame برای تبدیل برچسب های رشته به معادل های عددی استفاده میشود.

*# Load library*

import pandas as pd

*# Create features*

dataframe = pd.DataFrame({"Score": [ "Low", "Low", "Medium", "Medium", "High"] } )

*# Create mapper*

scale\_mapper = {"Low":1,

"Medium":2,

"High":3}

*# Replace feature values with scale*

dataframe["Score"].replace(scale\_mapper)

0 1

1 1

2 2

3 2

4 3

Name: Score, dtype: int64

**بحث**

اغلب ما یک ویژگی با کلاس هایی داریم که دارای نوعی طبیعی هستند یک مثال معروف مقیاس لیکرت است:

* کاملا موافقم
* موافق
* خنثی
* مخالف بودن
* به شدت مخالف

هنگام رمزگذاری ویژگی برای استفاده در یادگیری ماشین، باید کلاس‌های ترتیبی را به مقادیر عددی تبدیل کنیم که مفهوم مرتب‌سازی را حفظ کند. متداول ترین روش ایجاد دیکشنری است که برچسب رشته کلاس را به یک عدد ترسیم می کند و سپس آن نقشه را روی ویژگی اعمال می کند. مهم است که انتخاب مقادیر عددی ما بر اساس اطلاعات قبلی ما در مورد کلاس های ترتیبی باشد. در راه حل ما، بالا به معنای واقعی کلمه سه برابر بزرگتر از پایین است. این در هر موردی خوب است، اما اگر فواصل فرضی بین کلاس ها برابر نباشد، می تواند خراب شود:

dataframe = pd.DataFrame ( { "Score": [ "Low",

"Low",

"Medium",

"Medium",

"High",

"Barely More Than Medium" ] } )

scale\_mapper = {"Low":1,

"Medium":2,

"Barely More Than Medium": 3,

"High":4 }

Dataframe ["Score"].replace(scale\_mapper)

0 1

1 1

2 2

3 2

4 4

5 3

Name: Score, dtype : int64

در این مثال، فاصله بین Low و Medium برابر با فاصله بین Medium و Barely More Than Medium است که تقریباً مطمئناً دقیق نیست. بهترین رویکرد این است که نسبت به مقادیر عددی نگاشت شده به کلاس ها آگاه باشید:

scale\_mapper = {"Low":1,

"Medium":2,

"Barely More Than Medium": 2.1,

"High":3}

Dataframe ["Score"] .replace(scale\_mapper)

0 1.0

1 1.0

2 2.0

3 2.0

4 3.0

5 2.1

Name: Score, dtype: float64

**3.5 رمزگذاری دیکشنری ویژگی ها**

**مسئله**

شما یک فرهنگ لغت دارید و می خواهید آن را به یک ماتریس ویژگی تبدیل کنید.

**راه حل**

استفاده از DictVectorizer

*# Import library*

from sklearn.feature\_extraction import DictVectorizer

*# Create dictionary*

data\_dict = [{"Red": 2, "Blue": 4},

{"Red": 4, "Blue": 3},

{"Red": 1, "Yellow": 2},

{"Red": 2, "Yellow": 2}]

*# Create dictionary vectorizer*

dictvectorizer = DictVectorizer(sparse=False)

*# Convert dictionary to feature matrix*

features = dictvectorizer.fit\_transform(data\_dict)

*# View feature matrix*

features

Array ( [ [ 4., 2., 0.],

[ 3., 4., 0.],

[ 0., 1., 2.],

[ 0., 2., 2.] ] )

به طور پیش فرض DictVectorizer یک ماتریس پراکنده را خروجی می دهد که فقط عناصر با مقداری غیر از 0 را ذخیره می کند. هنگامی که ماتریس های عظیمی داریم (که اغلب در پردازش زبان طبیعی با آن مواجه می شویم) و می خواهیم نیازهای حافظه را به حداقل برسانیم، این می تواند بسیار مفید باشد. می‌توانیم DictVectorizer را مجبور کنیم که یک ماتریس متراکم را با استفاده از sparse=False تولید کند. با استفاده از روش get\_feature\_names می توانیم نام هر ویژگی تولید شده را بدست آوریم

*# Get feature names*

feature\_names = dictvectorizer.get\_feature\_names()

*# View feature names*

feature\_names

['Blue', 'Red', 'Yellow']

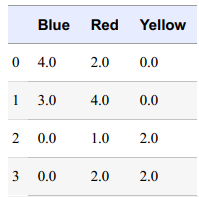
در حالی که ضروری نیست، برای مثال می توانیم یک DataFrame ایجاد کنیم تا خروجی را بهتر ببینیم::

*# Import library*

import pandas as pd

*# Create dataframe from features*

pd.DataFrame(features, columns=feature\_names)



**بحث**

دیکشنری یک ساختار داده محبوب است که توسط بسیاری از زبان های برنامه نویسی استفاده می شود. با این حال، الگوریتم‌های یادگیری ماشین انتظار دارند داده‌ها در قالب یک ماتریس باشند. ما می توانیم این کار را با استفاده از dictvectorizer انجام دهیم. این یک وضعیت رایج هنگام کار با پردازش زبان طبیعی است. برای مثال، ممکن است مجموعه‌ای از اسناد داشته باشیم و برای هر سند یک فرهنگ لغت داشته باشیم که حاوی تعداد دفعاتی است که هر کلمه در سند ظاهر می‌شود. با استفاده از dictvectorizer، ما به راحتی می توانیم یک ماتریس ویژگی ایجاد کنیم که در آن هر ویژگی تعداد دفعاتی است که یک کلمه در هر سند ظاهر می شود:

*# Create word counts dictionaries for four documents*

doc\_1\_word\_count = {"Red": 2, "Blue": 4}

doc\_2\_word\_count = {"Red": 4, "Blue": 3}

doc\_3\_word\_count = {"Red": 1, "Yellow": 2}

doc\_4\_word\_count = {"Red": 2, "Yellow": 2}

*# Create list*

doc\_word\_counts = [doc\_1\_word\_count,

doc\_2\_word\_count,

doc\_3\_word\_count,

doc\_4\_word\_count]

*# Convert list of word count dictionaries into feature matrix*

dictvectorizer.fit\_transform(doc\_word\_counts)

Array ( [ [ 4., 2., 0.],

[ 3., 4., 0.],

[ 0., 1., 2.],

[ 0., 2., 2.] ] )

در مثال اسباب بازی ما تنها سه کلمه منحصر به فرد (Red, Yellow, Blue) وجود دارد، بنابراین تنها سه ویژگی در ماتریس ما وجود دارد. با این حال، می‌توانید تصور کنید که اگر هر سند واقعاً یک کتاب در کتابخانه دانشگاه باشد، ماتریس ویژگی‌های ما بسیار بزرگ خواهد بود (و سپس می‌خواهیم ذخیره را روی True تنظیم کنیم).

همچنین ببینید

* [نحوه استفاده از دیکشنری ها در پایتون](http://bit.ly/2HReoWz)
* [ماتریس های پراکنده SciPy](http://bit.ly/2HReBZR)

**4.5 درج مقادیر گمشده کلاس**

**مسئله**

شما یک ویژگی دسته بندی دارید که حاوی مقادیر گم شده است می خواهید با مقادیر پیش بینی شده جایگزین کنید.

**راه حل**

راه حل ایده آل آموزش یک الگوریتم طبقه بندی کننده یادگیری ماشین است برای پیش‌بینی مقادیر گمشده، معمولاً k نزدیک‌ترین همسایه‌ها

طبقه بندی کننده (KNN):

*# Load libraries*

import numpy as np

from sklearn.neighbors import KNeighborsClassifier

*# Create feature matrix with categorical feature*

X = np.array( [ [0, 2.10, 1.45],

[1, 1.18, 1.33],

[0, 1.22, 1.27],

[1, -0.21, -1.19] ])

*# Create feature matrix with missing values in the categorical feature*

X\_with\_nan = np.array([[np.nan, 0.87, 1.31],

[np.nan, -0.67, -0.22] ] )

*# Train KNN learner*

clf = KNeighborsClassifier(3, weights='distance')

trained\_model = clf.fit( X[:,1:], X[:,0] )

*# Predict missing values' class*

imputed\_values = trained\_model.predict(X\_with\_nan[:,1:])

*# Join column of predicted class with their other features*

X\_with\_imputed = np.hstack((imputed\_values.reshape(-1,1),

X\_with\_nan[:,1:]))

*# Join two feature matrices*

np.vstack((X\_with\_imputed, X))

array ( [ [ 0. , 0.87, 1.31],

[ 1. , -0.67, -0.22],

[ 0. , 2.1 , 1.45],

[ 1. , 1.18, 1.33],

[ 0. , 1.22, 1.27],

[ 1. , -0.21, -1.19] ] )

یک راه حل جایگزین، پر کردن مقادیر از دست رفته با متداول ترین مقدار ویژگی است:

from sklearn.preprocessing import Imputer

*# Join the two feature matrices*

X\_complete = np.vstack( (X\_with\_nan, X) )

imputer = Imputer (strategy= 'most\_frequent', axis=0)

imputer.fit\_transform (X\_complete)

array ( [ [ 0. , 0.87, 1.31],

[ 0. , -0.67, -0.22],

[ 0. , 2.1 , 1.45],

[ 1. , 1.18, 1.33],

[ 0. , 1.22, 1.27],

[ 1. , -0.21, -1.19 ] ] )

**بحث**

وقتی مقادیر گمشده در یک ویژگی طبقه بندی شده داریم، بهترین راه حل ما این است که جعبه ابزار الگوریتم های یادگیری ماشین را برای پیش بینی مقادیر مشاهدات از دست رفته باز کنیم. ما می توانیم این کار را با در نظر گرفتن ویژگی با مقادیر از دست رفته به عنوان بردار هدف و سایر ویژگی ها به عنوان ماتریس ویژگی انجام دهیم. الگوریتمی که معمولاً مورد استفاده قرار می گیرد KNN است (که بعداً در این کتاب به طور عمیق مورد بحث قرار می گیرد)، که به مقدار گمشده کلاس میانه k نزدیکترین مشاهدات را نسبت می دهد. از طرف دیگر، می‌توانیم مقادیر از دست رفته را با متداول‌ترین کلاس ویژگی پر کنیم. اگرچه نسبت به KNN پیچیده تر است، اما برای داده های بزرگتر مقیاس پذیرتر است. در هر صورت، توصیه می‌شود که یک ویژگی باینری را شامل شود که نشان می‌دهد کدام مشاهدات حاوی مقادیر منتسب هستند.

**همچنین**

* [غلبه بر مقادیر از دست رفته در یک طبقه‌بندی جنگل تصادفی](http://bit.ly/2HSsNBF)
* [مطالعه K -نزدیکترین همسایه به عنوان یک روش انتساب](http://bit.ly/2HS9sAT)

**5.5 مدیریت کلاس های نامتعادل**

**مسئله**

شما یک بردار هدف با کلاس های بسیار نامتعادل دارید.

**راه حل**

داده های بیشتری جمع آوری کنید اگر این امکان پذیر نیست، معیارهای مورد استفاده برای ارزیابی مدل خود را تغییر دهید. اگر جواب نداد، از پارامترهای وزن کلاس داخلی مدل (در صورت وجود)، نمونه برداری پایین یا نمونه برداری استفاده کنید. ما معیارهای ارزیابی را در فصل بعدی پوشش می‌دهیم، بنابراین فعلاً اجازه دهید روی پارامترهای وزن کلاس، نمونه‌برداری پایین و نمونه‌برداری بالا تمرکز کنیم. برای نشان دادن راه حل های خود، باید داده هایی با کلاس های نامتعادل ایجاد کنیم.

مجموعه داده Fisher’sشامل سه کلاس متعادل از 50 مشاهدات است که هر کدام گونه‌های گل را نشان می‌دهند (Iris setosa، Iris virginica و Iris versicolor). برای عدم تعادل مجموعه داده، 40 مورد از 50 مشاهدات Iris setosa را حذف می کنیم و سپس کلاس Iris virginica و Iris versicolor را ادغام می کنیم. نتیجه نهایی یک بردار هدف باینری است که نشان می دهد آیا مشاهده گل زنبق ستوزا است یا نه. نتیجه 10 مشاهده از Iris setosa (کلاس 0) و 100 مشاهده از Iris setosa ( کلاس 1) است:

*# Load libraries*

import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import load\_iris

*# Load iris data*

iris = load\_iris()

*# Create feature matrix*

features = iris.data

# *Create target vector*

target = iris.target

*# Remove first 40 observations*

features = features[40:,:]

target = target[40:]

*# Create binary target vector indicating if class 0*

target = np.where((target == 0), 0, 1)

*# Look at the imbalanced target vector*

target

Array ( [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1 , 1, 1, 1 , 1,

1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] )

بسیاری از الگوریتم‌ها در Sikit-Learn پارامتری را به کلاس‌های وزنی در طول تمرین ارائه می‌کنند تا تأثیر عدم تعادل آنها را خنثی کنند. در حالی که ما هنوز آن را پوشش نداده‌ایم، RandomForestClassifier یک الگوریتم طبقه‌بندی محبوب است و شامل یک پارامتر class\_weight است. شما می توانید یک آرگومان ارسال کنید که وزن کلاس مورد نظر را به صراحت مشخص کند:

*# Create weights*

weights = {0: .9, 1: 0.1}

*# Create random forest classifier with weights*

RandomForestClassifier(class\_weight=weights)

RandomForestClassifier(bootstrap=True, class\_weight={0: 0.9, 1: 0.1},

criterion='gini', max\_depth=None,

max\_features = 'auto',

max\_leaf\_nodes=None, min\_impurity\_decrease=0.0,

min\_impurity\_split=None, min\_samples\_leaf=1,

min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0,

n\_estimators=10, n\_jobs=1, oob\_score=False,

random\_state= None,

verbose=0, warm\_start=False)

یا می توانید به صورت متعادل پاس دهید، که به طور خودکار وزن هایی را به طور معکوس با فرکانس های کلاس ایجاد می کند:

*# Train a random forest with balanced class weights*

RandomForestClassifier(class\_weight="balanced")

RandomForestClassifier(bootstrap=True,

class\_weight = 'balanced',

criterion='gini', max\_depth = None,

max\_features = 'auto',

max\_leaf\_nodes =None, min\_impurity\_decrease =0.0,

min\_impurity\_split =None, min\_samples\_leaf =1,

min\_samples\_split =2,

min\_weight\_fraction\_leaf =0.0,

n\_estimators=10, n\_jobs =1, oob\_score = False,

random\_state = None,

verbose=0, warm\_start = False)

از طرف دیگر، می‌توانیم کلاس اکثریت را پایین‌نمونه‌سازی کنیم یا کلاس اقلیت را نمونه‌برداری کنیم. در نمونه‌گیری پایین، ما به‌طور تصادفی بدون جایگزینی از کلاس اکثریت (یعنی کلاس با مشاهدات بیشتر) نمونه‌برداری می‌کنیم تا یک زیرمجموعه جدید از مشاهدات برابر با کلاس اقلیت ایجاد کنیم. به عنوان مثال، اگر کلاس اقلیت 10 مشاهده داشته باشد، به طور تصادفی 10 مشاهده را از طبقه اکثریت انتخاب می کنیم و از آن 20 مشاهده به عنوان داده استفاده می کنیم. در اینجا ما دقیقاً این کار را با استفاده از داده های Iris نامتعادل خود انجام می دهیم:

*# Indicies of each class' observations*

i\_class0 = np.where(target == 0)[0]

i\_class1 = np.where(target == 1)[0]

*# Number of observations in each class*

n\_class0 = len(i\_class0)

n\_class1 = len(i\_class1)

*# For every observation of class 0, randomly sample*

*# from class 1 without replacement*

i\_class1\_downsampled = np.random.choice(i\_class1,

size=n\_class0, replace=False)

*# Join together class 0's target vector with the*

*# downsampled class 1's target vector*

np.hstack((target[i\_class0], target[i\_class1\_downsampled]))

array ( [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1 ] )

*# Join together class 0's feature matrix with the*

*# downsampled class 1's feature matrix*

np.vstack( (features[i\_class0,:],

features[i\_class1\_downsampled,:] ) ) [0:5]

array ( [ [ 5. , 3.5, 1.3, 0.3],

[ 4.5, 2.3, 1.3, 0.3],

[ 4.4, 3.2, 1.3, 0.2],

[ 5. , 3.5, 1.6, 0.6],

[ 5.1, 3.8, 1.9, 0.4] ] )

گزینه دیگر ما این است که از کلاس اقلیت نمونه برداری کنیم. در نمونه برداری، برای هر مشاهده در کلاس اکثریت، به طور تصادفی یک مشاهده از کلاس اقلیت را با جایگزینی انتخاب می کنیم. نتیجه نهایی همان تعداد مشاهدات طبقات اقلیت و اکثریت است.

Upsampling بسیار شبیه به downsampling اجرا می شود، فقط به صورت معکوس:

*# For every observation in class 1, randomly sample from class 0 with replacement*

i\_class0\_upsampled = np.random.choice(i\_class0,

size=n\_class1, replace=True)

*# Join together class 0's upsampled target vector with class*

*1's target vector*

np.concatenate((target[i\_class0\_upsampled],

target[i\_class1]))

array ( [0, 0 , 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0 , 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0 , 0, 0, 0,

0, 0, 0, 0, 0 , 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1,

1, 1 , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1 , 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1,

1, 1, 1, 1 , 1 , 1, 1, 1, 1, 1, 1, 1 , 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] )

*# Join together class 0's upsampled feature matrix with*

*class 1's feature matrix*

np.vstack((features[i\_class0\_upsampled,:],

features[i\_class1,:]))[0:5]

array ( [ [ 5. , 3.5, 1.6, 0.6],

[ 5. , 3.5, 1.6, 0.6],

[ 5. , 3.3, 1.4, 0.2],

[ 4.5, 2.3, 1.3, 0.3],

[ 4.8, 3. , 1.4, 0.3] ] )

**بحث**

در دنیای واقعی هم کلاس‌های نامتعادل در همه جا وجود دارد. بیشتر بازدیدکنندگان روی دکمه خرید کلیک نمی‌کنند و بسیاری از انواع سرطان ها خوشبختانه نادر هستند. به همین دلیل، مدیریت کلاس­های نامتعادل یک فعالیت رایج در یادگیری ماشین است. بهترین استراتژی ما صرفاً جمع آوری مشاهدات بیشتر به ویژه مشاهدات طبقه اقلیت است. ولی این اغلب ممکن نیست، بنابراین ما باید به گزینه های دیگر متوسل شویم.

راهبرد دوم استفاده از معیار ارزیابی مدل است که برای کلاس‌های نامتعادل مناسب‌تر است. دقت، اغلب به‌عنوان معیاری برای ارزیابی عملکرد یک مدل استفاده می‌شود، اما وقتی کلاس‌های نامتعادل وجود دارند، دقت می‌تواند مناسب باشد. به عنوان مثال، اگر فقط 0.5٪ از مشاهدات سرطان از نوع نادر داشته باشند، حتی یک مدل ساده که پیش بینی می­کند هیچ کس سرطان ندارد، 99.5٪ دقیق خواهد بود. واضح است که این ایده آل نیست. برخی از معیارهای دیگر مانند ماتریس های درهم ریختگی، دقت، صحت، امتیازات F1 و منحنی های ROC در فصل های بعدی مورد بحث قرار می دهیم.

استراتژی سوم استفاده از پارامترهای وزن کلاس موجود در آن است. پیاده سازی برخی از مدل ها این به ما این امکان می­دهد که الگوریتم را برای کلاس­های نامتعادل تنظیم کنیم. خوشبختانه، بسیاری از طبقه‌بندی‌کننده‌های scikitlearn دارای پارامتر class\_weight هستند که آن را به گزینه خوبی تبدیل کرده است.

راهبردهای چهارم و پنجم یعنی نمونه برداری پایین و نمونه برداری بالا مرتبط هستند. در نمونه برداری پایین، یک زیرمجموعه تصادفی از کلاس اکثریت با اندازه مساوی با کلاس اقلیت ایجاد می­کنیم. در نمونه‌برداری بالا، ما به طور مکرر با جایگزینی از کلاس اقلیت نمونه‌برداری می‌کنیم تا اندازه آن با کلاس اکثریت برابر باشد. تصمیم بین استفاده از نمونه‌برداری پایین و نمونه‌برداری بالا به یک زمینه خاص بستگی دارد، و به طور کلی باید هر دو را امتحان کنیم تا ببینیم کدامیک نتایج بهتری ایجاد می‌کند.

1. Categorical data [↑](#footnote-ref-1)
2. nominal [↑](#footnote-ref-2)
3. ordinal [↑](#footnote-ref-3)