支持向量机SVM案例：预测明天是否会下雨

一、概述

SVC在现实中的应用十分广泛，尤其实在图像和文字识别方面。然而，这些数据不仅非常难以获取，还难以在课程中完整呈现出来，但SVC真实应用的代码其实就是sklearn中的三行，真正能够展现出SVM强大之处的，反而很少是案例本身，而是我们之前所作的各种探索。

我们在学习算法的时候，会使用各种各样的数据集来进行演示，但这些数据往往非常干净并且规整，不需要做太多的数据预处理。在实际工作中，数据预处理往往比建模难得多，耗时多得多，因此合理的数据预处理是非常必要的。考虑到大家渴望学习真实数据上的预处理的需求，以及SVM需要在比较规则的数据集上来表现的特性，我为大家准备了这个Kaggle上下载的，未经过预处理的澳大利亚天气数据集。我们的目标是在这个数据集上来预测明天是否会下雨。

这个案例的核心目的，是通过巧妙的预处理和特征工程来向大家展示，在现实数据集上我们往往如何做数据预处理，或者我们都有哪些预处理的方式和思路。预测天气是一个非常非常困难的主题，因为影响天气的因素太多，而Kaggle的这份数据也丝毫不让我们失望，是一份非常难的数据集，难到我们目前学过的所有算法在这个数据集上都不会有太好的结果，尤其是召回率recall，异常地低。在这里，我为大家抛砖引玉，在这个15W行数据的数据集上，随机抽样5000个样本来为大家演示我的数据预处理和特征工程的过程，为大家提供一些数据预处理和特征工程的思路。不过，特征工程没有标准答案，因此大家应当多尝试，希望使用原数据集的小伙伴们可以到Kaggle下载最原始版本，或者直接从我们的课件打包下载的数据中获取：

Kaggle下载链接走这里：https://www.kaggle.com/jsphyg/weather-dataset-rattle-package

对于使用Kaggle原数据集的小伙伴的温馨提示：
记得好好阅读Kaggle上的各种数据集说明哦~！有一些特征是不能够使用的！

那就让我们开始我们的案例吧。

二、数据预处理

1. 导库导数据，探索特征

导入需要的库

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

导入数据，探索数据

```
df = pd.read_csv("weather.csv",index_col=0)
df.head()

df.shape   #(142193, 22)

#抽取5000条样本进行后续处理
weather = df.sample(n=5000,random_state=0)
weather.index = range(weather.shape[0])
weather.head()
```

来查看一下各个特征都代表了什么：

特征/标签	含义
Date	观察日期
Location	获取该信息的气象站的名称
MinTemp	以摄氏度为单位的最低温度
MaxTemp	以摄氏度为单位的最高温度
Rainfall	当天记录的降雨量，单位为mm
Evaporation	到早上9点之前的24小时的A级蒸发量（mm）
Sunshine	白日受到日照的完整小时
WindGustDir	在到午夜12点前的24小时中的最强风的风向
WindGustSpeed	在到午夜12点前的24小时中的最强风速（km / h）
WindDir9am	上午9点时的风向
WindDir3pm	下午3点时的风向
WindSpeed9am	上午9点之前每个十分钟的风速的平均值（km / h）
WindSpeed3pm	下午3点之前每个十分钟的风速的平均值（km / h）
Humidity9am	上午9点的湿度（百分比）
Humidity3am	下午3点的湿度（百分比）
Pressure9am	上午9点平均海平面上的大气压（hpa）
Pressure3pm	下午3点平均海平面上的大气压（hpa）
Cloud9am	上午9点的天空被云层遮蔽的程度，这是以“oktas”来衡量的，这个单位记录了云层遮挡天空的程度。0表示完全晴朗的天空，而8表示它完全是阴天。
Cloud3pm	下午3点的天空被云层遮蔽的程度
Temp9am	上午9点的摄氏度温度
Temp3pm	下午3点的摄氏度温度
RainTomorrow	目标变量，我们的标签：明天下雨了吗?

```
#将特征矩阵和标签Y分开
X = weather.iloc[:,:-1]
Y = weather.iloc[:,-1]

#探索数据类型
X.info()

#探索缺失值
X.isnull().mean()

#探索标签的分类
np.unique(Y)
```

粗略观察可以发现，这个特征矩阵由一部分分类变量和一部分连续变量组成，其中云层遮蔽程度虽然是以数字表示，但是本质却是分类变量。大多数特征都是采集的自然数据，比如蒸发量，日照时间，湿度等等，而少部分特征是人为构成的。还有一些是单纯表示样本信息的变量，比如采集信息的地点，以及采集的时间。

2. 分数据集，优先探索标签

分训练集和测试集，并做描述性统计

```
#分训练集和测试集
Xtrain, Xtest, Ytrain, Ytest =
train_test_split(X,Y,test_size=0.3,random_state=420)

#恢复索引
for i in [Xtrain, Xtest, Ytrain, Ytest]:
    i.index = range(i.shape[0])
```

在现实中，我们会先分训练集和测试集，再开始进行数据预处理。这是由于，测试集在现实中往往是不可获得的，或者被假设为是不可获得的，我们不希望我们建模的任何过程受到测试集数据的影响，否则的话，就相当于提前告诉了模型一部分预测的答案。在这里，为了让案例尽量接近真实的样貌，所以采取了现实中所使用的这种方式：先分训练集和测试集，再一步步进行预处理。这样导致的结果是，我们对训练集执行的所有操作，都必须对测试集执行一次，工作量是翻倍的。

```
#是否有样本不平衡问题?
Ytrain.value_counts()
Ytest.value_counts()

#将标签编码
from sklearn.preprocessing import LabelEncoder
encorder = LabelEncoder().fit(Ytrain)
Ytrain = pd.DataFrame(encorder.transform(Ytrain))
Ytest = pd.DataFrame(encorder.transform(Ytest))
```

3. 探索特征，开始处理特征矩阵

3.1 描述性统计与异常值

```
#描述性统计
Xtrain.describe([0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.99]).T
Xtest.describe([0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.99]).T

"""
对于去kaggle上下载了数据的童鞋们，以及坚持要使用完整版数据的（15w行）童鞋们
如果你发现了异常值，首先你要观察，这个异常值出现的频率
如果异常值只出现了一次，多半是输入错误，直接把异常值删除
如果异常值出现了多次，去跟业务人员沟通，可能这是某种特殊表示，如果是人为造成的错误，异常值留着是没有用的，只要数据量不是太大，都可以删除
如果异常值占到你总数据量的10%以上了，不能轻易删除。可以考虑把异常值替换成非异常但是非干扰的项，比如说用0来进行替换，或者把异常当缺失值，用均值或者众数来进行替换
"""

#先查看原始的数据结构
Xtrain.shape
Xtest.shape

#提取出所有的数值型特征的列名
col = Xtrain.mean().index
#查看训练集各列异常值的比例
(np.abs((Xtrain.loc[:,col] -
Xtrain.mean())/Xtrain.std())>3).sum()/Xtrain.shape[0]

#少数存在，于是采取删除的策略
#注意如果删除特征矩阵，则必须连对应的标签一起删除，特征矩阵的行和标签的行必须要一一对应
#提取出所有训练集所有异常值的索引
deltrain = []
for i in col:
    bool_ = np.abs((Xtrain.loc[:,i] -
Xtrain.loc[:,i].mean())/Xtrain.loc[:,i].std())>3
    ind = Xtrain[bool_].index
    deltrain.extend(list(ind))

#去重之后有异常值的行索引
deltrain1 = list(set(deltrain))

len(deltrain1)

#查看测试集各列缺失值（注意这里的均值和方差需要用训练集的）
(np.abs((Xtest.loc[:,col] -
Xtrain.mean())/Xtrain.std())>3).sum()/Xtest.shape[0]

#提取出测试集的所有异常值的索引
```

```
deltest = []
for i in col:
    bool_ = np.abs((Xtest.loc[:,i] -
Xtrain.loc[:,i].mean())/Xtrain.loc[:,i].std())>3
    ind = Xtest[bool_].index
    deltest.extend(list(ind))

#去重之后有异常值的行索引
deltest1 = list(set(deltest))

len(deltest1)

#删除异常值（特征和标签都要删除）
Xtrain = Xtrain.drop(index=deltrain1)
Ytrain = Ytrain.drop(index=deltrain1)
Xtest = Xtest.drop(index=deltest1)
Ytest = Ytest.drop(index=deltest1)

#删除完毕之后，观察原始的数据结构，确认删除正确
Xtrain.shape

#进行任何行删除之后，千万记得要恢复索引
for i in [Xtrain, Xtest, Ytrain, Ytest]:
    i.index = range(i.shape[0])

Xtrain.head()
Xtest.head()
```

3.2 处理困难特征：日期

我们采集数据的日期是否和我们的天气有关系呢？我们可以探索一下我们的采集日期有什么样的性质：

```
Xtrainc = Xtrain.copy()

Xtrain.iloc[:,0].value_counts()
#首先，日期不是独一无二的，日期有重复
#其次，在我们分训练集和测试集之后，日期也不是连续的，而是分散的
#某一年的某一天倾向于会下雨？或者倾向于不会下雨吗？
#不是日期影响了下雨与否，反而更多的是这一天的日照时间，湿度，温度等等这些因素影响了是否会下雨
#光看日期，其实感觉它对我们的判断并无直接影响
#如果我们把它当作连续型变量处理，那算法会人为它是一系列1~3000左右的数字，不会意识到这是日期

Xtrain.iloc[:,0].value_counts().count()
#如果我们把它当作分类型变量处理，类别太多，有2025类，如果换成数值型，会被直接当成连续型变量，
如果做成哑变量，我们特征的维度会爆炸
```

如果我们的思考简单一些，我们可以直接删除日期这个特征。首先它不是一个直接影响我们标签的特征，并且要处理日期其实是非常困难的。如果大家认可这种思路，那可以直接运行下面的代码来删除我们的日期：

```
Xtrain = Xtrain.drop(["Date"],axis=1)
Xtest = Xtest.drop(["Date"],axis=1)
```

但在这里，很多人可能会持不同意见，怎么能够随便删除一个特征（哪怕我们已经觉得它可能无关）？如果我们要删除，我们可能需要一些统计过程，来判断说这个特征确实是和标签无关的，那我们可以先将“日期”这个特征编码后对它和标签做方差齐性检验（ANOVA），如果检验结果表示日期这个特征的确和我们的标签无关，那我们就可以愉快地删除这个特征了。但要编码“日期”这个特征，就又回到了它到底是否会被算法当成是分类变量的问题上。

其实我们可以想到，日期必然是和我们的结果有关的，它会从两个角度来影响我们的标签：

首先，我们可以想到，昨天的天气可能会影响今天的天气，而今天的天气又可能会影响明天的天气。也就是说，随着日期的逐渐改变，样本是会受到上一个样本的影响的。但是对于算法来说，普通的算法是无法捕捉到样本与样本之间的联系的，我们的算法捕捉的是样本的每个特征与标签之间的联系（即列与列之间的联系），而无法捕捉样本与样本之间的联系（行与行的联系）。

要让算法理解上一个样本的标签可能会影响下一个样本的标签，我们必须使用时间序列分析。时间序列分析是指将同一统计指标的数值按其发生的时间先后顺序排列而成的数列。时间序列分析的主要目的是根据已有的历史数据对未来进行预测。然而，（据我所知）时间序列只能在单调的，唯一的时间上运行，即一次只能够对一个地点进行预测，不能够实现一次性预测多个地点，除非进行循环。而我们的时间数据本身，不是单调的，也不是唯一的，经过抽样之后，甚至连连续的都不是了，我们的时间是每个混杂在多个地点中，每个地点上的一小段时间。如何使用时间序列来处理这个问题，就会变得复杂。

那我们可以换一种思路，既然算法处理的是列与列之间的关系，我是否可以把”今天的天气会影响明天的天气“这个指标转换成一个特征呢？我们就这样来操作。

我们观察到，我们的特征中有一列叫做“Rainfall"，这是表示当前日期当前地区下的降雨量，换句话说，也就是”今天的降雨量“。凭常识我们认为，今天是否下雨，应该会影响明天是否下雨，比如有的地方可能就有这样的气候，一旦下雨就连着下很多天，也有可能有的地方的气候就是一场暴雨来得快去的快。因此，我们可以将时间对气候的连续影响，转换为”今天是否下雨“这个特征，巧妙地将样本对应标签之间的联系，转换成是特征与标签之间的联系了。

```
Xtrain["Rainfall"].head(20)

Xtrain.loc[Xtrain["Rainfall"] >= 1,"RainToday"] = "Yes"
Xtrain.loc[Xtrain["Rainfall"] < 1,"RainToday"] = "No"
Xtrain.loc[Xtrain["Rainfall"] == np.nan,"RainToday"] = np.nan

Xtest.loc[Xtest["Rainfall"] >= 1,"RainToday"] = "Yes"
Xtest.loc[Xtest["Rainfall"] < 1,"RainToday"] = "No"
Xtest.loc[Xtest["Rainfall"] == np.nan,"RainToday"] = np.nan

Xtrain.head()
Xtest.head()
```

如此，我们就创造了一个特征，今天是否下雨“RainToday”。

那现在，我们是否就可以将日期删除了呢？对于我们而言，日期本身并不影响天气，但是日期所在的月份和季节其实是影响天气的，如果任选梅雨季节的某一天，那明天下雨的可能性必然比非梅雨季节的那一天要大。虽然我们无法让机器学习体会不同月份是什么季节，但是我们可以对不同月份进行分组，算法可以通过训练感受到，“这个月或者这个季节更容易下雨”。因此，我们可以将月份或者季节提取出来，作为一个特征使用，而舍弃掉具体的日期。如此，我们又可以创造第二个特征，月份"Month"。

```
int(Xtrain.loc[0,"Date"].split("-")[1]) #提取出月份

Xtrain["Date"] = Xtrain["Date"].apply(lambda x:int(x.split("-")[1]))
#替换完毕后，我们需要修改列的名称
#rename是比较少有的，可以用来修改单个列名的函数
#我们通常都直接使用 df.columns = 某个列表 这样的形式来一次修改所有的列名
#但rename允许我们只修改某个单独的列
Xtrain = Xtrain.rename(columns={"Date":"Month"})

Xtrain.head()

Xtest["Date"] = Xtest["Date"].apply(lambda x:int(x.split("-")[1]))
Xtest = Xtest.rename(columns={"Date":"Month"})

Xtest.head()
```

通过时间，我们处理出两个新特征，“今天是否下雨”和“月份”。

3.3 处理分类型变量：缺失值

接下来，我们总算可以开始处理我们的缺失值了。首先我们要注意到，由于我们的特征矩阵由两种类型的数据组成：分类型和连续型，因此我们必须对两种数据采用不同的填补缺失值策略。传统地，如果是分类型特征，我们则采用众数进行填补。如果是连续型特征，我们则采用均值来填补。

此时，由于我们已经分了训练集和测试集，我们需要考虑一件事：究竟使用哪一部分的数据进行众数填补呢？答案是，使用训练集上的众数对训练集和测试集都进行填补。为什么会这样呢？按道理说就算用测试集上的众数对测试集进行填补，也不会使测试集数据进入我们建好的模型，不会给模型透露一些信息。然而，在现实中，我们的测试集未必是很多条数据，也许我们的测试集只有一条数据，而某个特征上是空值，此时此刻测试集本身的众数根本不存在，要如何利用测试集本身的众数去进行填补呢？因此为了避免这种尴尬的情况发生，我们假设测试集和训练集的数据分布和性质都是相似的，因此我们统一使用训练集的众数和均值来对测试集进行填补。

在sklearn当中，即便是我们的填补缺失值的类也需要由实例化，fit和接口调用执行填补三个步骤来进行，而这种分割其实一部分也是为了满足我们使用训练集的建模结果来填补测试集的需求。我们只需要实例化后，使用训练集进行fit，然后在调用接口执行填补时用训练集fit后的结果分别来填补测试集和训练集就可以了。

```
#查看缺失值的缺失情况
Xtrain.isnull().mean()

#首先找出，分类型特征都有哪些
cate = Xtrain.columns[Xtrain.dtypes == "object"].tolist()
```

```
#除了特征类型为"object"的特征们，还有虽然用数字表示，但是本质为分类型特征的云层遮蔽程度
cloud = ["Cloud9am","Cloud3pm"]
cate = cate + cloud
cate

#对于分类型特征，我们使用众数来进行填补
from sklearn.impute import SimpleImputer

si = SimpleImputer(missing_values=np.nan,strategy="most_frequent")
#注意，我们使用训练集数据来训练我们的填补器，本质是在生成训练集中的众数
si.fit(Xtrain.loc[:,cate])

#然后我们用训练集中的众数来同时填补训练集和测试集
Xtrain.loc[:,cate] = si.transform(Xtrain.loc[:,cate])
Xtest.loc[:,cate] = si.transform(Xtest.loc[:,cate])

Xtrain.head()
Xtest.head()

#查看分类型特征是否依然存在缺失值
Xtrain.loc[:,cate].isnull().mean()
Xtest.loc[:,cate].isnull().mean()
```

3.4 处理分类型变量：将分类型变量编码

在编码中，和我们的填补缺失值一样，我们也是需要先用训练集fit模型，本质是将训练集中已经存在的类别转换成是数字，然后我们再使用接口transform分别在测试集和训练集上来编码我们的特征矩阵。当我们使用接口在测试集上进行编码的时候，如果测试集上出现了训练集中从未出现过的类别，那代码就会报错，表示说“我没有见过这个类别，我无法对这个类别进行编码”，此时此刻你就要思考，你的测试集上或许存在异常值，错误值，或者的确有一个新的类别出现了，而你曾经的训练数据中并没有这个类别。以此为基础，你需要调整你的模型。

```
#将所有的分类型变量编码为数字，一个类别是一个数字
from sklearn.preprocessing import OrdinalEncoder
oe = OrdinalEncoder()

#利用训练集进行fit
oe = oe.fit(Xtrain.loc[:,cate])

#用训练集的编码结果来编码训练和测试特征矩阵
#在这里如果测试特征矩阵报错，就说明测试集中出现了训练集中从未见过的类别
Xtrain.loc[:,cate] = oe.transform(Xtrain.loc[:,cate])
Xtest.loc[:,cate] = oe.transform(Xtest.loc[:,cate])

Xtrain.loc[:,cate].head()
Xtest.loc[:,cate].head()
```

3.5 处理连续型变量：填补缺失值

连续型变量的缺失值由均值来进行填补。连续型变量往往已经是数字，无需进行编码转换。与分类型变量中一样，我们也是使用训练集上的均值对测试集进行填补。如果学过随机森林填补缺失值的小伙伴，可能此时会问，为什么不使用算法来进行填补呢？使用算法进行填补也是没有问题的，但在现实中，其实我们非常少用到算法来进行填补，有以下几个理由：

1. 算法是黑箱，解释性不强。如果你是一个数据挖掘工程师，你使用算法来填补缺失值后，你不懂机器学习的老板或者同事问你的缺失值是怎么来的，你可能需要从头到尾帮他/她把随机森林解释一遍，这种效率过低的事情是不可能做的，而许多老板和上级不会接受他们无法理解的东西。
2. 算法填补太过缓慢，运行一次森林需要有至少100棵树才能够基本保证森林的稳定性，而填补一个列就需要很长的时间。在我们并不知道森林的填补结果是好是坏的情况下，填补一个很大的数据集风险非常高，有可能需要跑好几个小时，但填补出来的结果却不怎么优秀，这明显是一个低效的方法。

因此在现实工作时，我们往往使用易于理解的均值或者中位数来进行填补。当然了，在算法比赛中，我们可以穷尽一切我们能够想到的办法来填补缺失值以追求让模型的效果更好，不过现实中，除了模型效果之外，我们还要追求可解释性。

```
col = Xtrain.columns.tolist()

for i in cate:
    col.remove(i)

col

#实例化模型，填补策略为"mean"表示均值
impmean = SimpleImputer(missing_values=np.nan,strategy = "mean")
#用训练集来fit模型
impmean = impmean.fit(Xtrain.loc[:,col])
#分别在训练集和测试集上进行均值填补
Xtrain.loc[:,col] = impmean.transform(Xtrain.loc[:,col])
Xtest.loc[:,col] = impmean.transform(Xtest.loc[:,col])

Xtrain.head()
Xtest.head()
```

3.6 处理连续型变量：无量纲化

数据的无量纲化是SVM执行前的重要步骤，因此我们需要对数据进行无量纲化。但注意，这个操作我们不对分类型变量进行。

```
col.remove("Month")
col

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss = ss.fit(Xtrain.loc[:,col])
Xtrain.loc[:,col] = ss.transform(Xtrain.loc[:,col])
Xtest.loc[:,col] = ss.transform(Xtest.loc[:,col])

Xtrain.head()
Xtest.head()
```

特征工程到这里就全部结束了。大家可以分别查看一下我们的Ytrain，Ytest，Xtrain，Xtest，确保我们熟悉他们的结构并且确保我们的确已经处理完毕全部的内容。将数据处理完毕之后，建议大家都使用to_csv来保存我们已经处理好的数据集，避免我们在后续建模过程中出现覆盖了原有的数据集的失误后，需要从头开始做数据预处理。在开始建模之前，无比保存好处理好的数据，然后在建模的时候，重新将数据导入。

三、建模与模型评估

```
from time import time
import datetime
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.metrics import roc_auc_score, recall_score

Ytrain = Ytrain.iloc[:,0].ravel()
Ytest = Ytest.iloc[:,0].ravel()

#建模选择自然是我们的支持向量机SVC，首先用核函数的学习曲线来选择核函数
#我们希望同时观察，精确性，recall以及AUC分数
#因为SVM是计算量很大的模型，所以我们需要时刻监控我们的模型运行时间

for kernel in ["linear","poly","rbf","sigmoid"]:
    times = time()
    clf = SVC(kernel = kernel
              ,gamma="auto"
              ,degree = 1
             ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest,Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
```

```
    print("%s 's testing accuracy %f, recall is %f', auc is %f" %
(kernel,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

我们注意到，模型的准确度和auc面积还是勉勉强强，但是每个核函数下的recall都不太高。相比之下，其实线性模型的效果是最好的。那现在我们可以开始考虑了，在这种状况下，我们要向着什么方向进行调参呢？我们最想要的是什么？

我们可以有不同的目标：

一，我希望不计一切代价判断出少数类，得到最高的recall。

二，我们希望追求最高的预测准确率，一切目的都是为了让accuracy更高，我们不在意recall或者AUC。

三，我们希望达到recall，ROC和accuracy之间的平衡，不追求任何一个也不牺牲任何一个。

四、模型调参

1. 最求最高Recall

如果我们想要的是最高的recall，可以牺牲我们准确度，希望不计一切代价来捕获少数类，那我们首先可以打开我们的class_weight参数，使用balanced模式来调节我们的recall：

```
for kernel in ["linear","poly","rbf","sigmoid"]:
    times = time()
    clf = SVC(kernel = kernel
              ,gamma="auto"
              ,degree = 1
              ,class_weight = "balanced"
             ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest,Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
    print("%s 's testing accuracy %f, recall is %f', auc is %f" %
(kernel,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

在锁定了线性核函数之后，我甚至可以将class_weight调节得更加倾向于少数类，来不计代价提升recall。

```
times = time()
clf = SVC(kernel = "linear"
          ,gamma="auto"
          ,class_weight = {1:10} #注意，这里写的其实是，类别1：10，隐藏了类别0：1这个比例
         ).fit(Xtrain, Ytrain)
result = clf.predict(Xtest)
score = clf.score(Xtest,Ytest)
recall = recall_score(Ytest, result)
auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
print("testing accuracy %f, recall is %f', auc is %f" %(score,recall,auc))
print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

随着recall地无节制上升，我们的精确度下降得十分厉害，不过看起来AUC面积却还好，稳定保持在0.86左右。如果此时我们的目的就是追求一个比较高的AUC分数和比较好的recall，那我们的模型此时就算是很不错了。虽然现在，我们的精确度很低，但是我们的确精准地捕捉出了每一个雨天。

2. 追求最高准确率

在我们现有的目标（判断明天是否会下雨）下，追求最高准确率而不顾recall其实意义不大， 但出于练习的目的，我们来看看我们能够有怎样的思路。此时此刻我们不在意我们的Recall了，那我们首先要观察一下，我们的样本不均衡状况。如果我们的样本非常不均衡，但是此时却有很多多数类被判错的话，那我们可以让模型任性地把所有地样本都判断为0，完全不顾少数类。

```
valuec = pd.Series(Ytest).value_counts()
valuec

valuec[0]/valuec.sum()
```

初步判断，可以认为我们其实已经将大部分的多数类判断正确了，所以才能够得到现在的正确率。为了证明我们的判断，我们可以使用混淆矩阵来计算我们的特异度，如果特异度非常高，则证明多数类上已经很难被操作了。

```
#查看模型的特异度
from sklearn.metrics import confusion_matrix as CM
clf = SVC(kernel = "linear"
          ,gamma="auto"
         ).fit(Xtrain, Ytrain)
result = clf.predict(Xtest)

cm = CM(Ytest,result,labels=(1,0))
cm

specificity = cm[1,1]/cm[1,:].sum()
specificity #几乎所有的0都被判断正确了，还有不少1也被判断正确了
```

可以看到，特异度非常高，此时此刻如果要求模型将所有的类都判断为0，则已经被判断正确的少数类会被误伤，整体的准确率一定会下降。而如果我们希望通过让模型捕捉更多少数类来提升精确率的话，却无法实现，因为一旦我们让模型更加倾向于少数类，就会有更多的多数类被判错。

可以试试看使用class_weight将模型向少数类的方向稍微调整，已查看我们是否有更多的空间来提升我们的准确率。如果在轻微向少数类方向调整过程中，出现了更高的准确率，则说明模型还没有到极限。

```
irange = np.linspace(0.01,0.05,10)

for i in irange:
    times = time()
    clf = SVC(kernel = "linear"
              ,gamma="auto"
              ,class_weight = {1:1+i}
             ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest,Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
    print("under ratio 1:%f testing accuracy %f, recall is %f', auc is %f" %
(1+i,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

惊喜出现了，我们的最高准确度是82.97%，超过了我们之前什么都不做的时候得到的82.83%。可见，模型还是有潜力的。我们可以继续细化我们的学习曲线来进行调整：

```
irange_ = np.linspace(0.03,0.04,10)
for i in irange_:
    times = time()
    clf = SVC(kernel = "linear"
              ,gamma="auto"
              ,class_weight = {1:1+i}
              ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest,Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
    print("under ratio 1:%f testing accuracy %f, recall is %f', auc is %f" %
(1+i,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

模型的效果没有太好，并没有再出现比我们的82.97%精确度更高的取值。可见，模型在不做样本平衡的情况下，准确度其实已经非常接近极限了，让模型向着少数类的方向调节，不能够达到质变。如果我们真的希望再提升准确度，只能选择更换模型的方式，调整参数已经不能够帮助我们了。想想看什么模型在线性数据上表现最好呢?

```
from sklearn.linear_model import LogisticRegression as LR

logclf = LR(solver="liblinear").fit(Xtrain, Ytrain)
logclf.score(Xtest,Ytest)

C_range = np.linspace(3,5,10)

for C in C_range:
    logclf = LR(solver="liblinear",C=C).fit(Xtrain, Ytrain)
    print(C,logclf.score(Xtest,Ytest))
```

尽管我们实现了非常小的提升，但可以看出来，模型的精确度还是没有能够实现质变。也许，要将模型的精确度提升到90%以上，我们需要集成算法：比如，梯度提升树。大家如果感兴趣，可以自己下去试试看。

3. 追求平衡

我们前面经历了多种尝试，选定了线性核，并发现调节class_weight并不能够使我们模型有较大的改善。现在我们来试试看调节线性核函数的C值能否有效果：

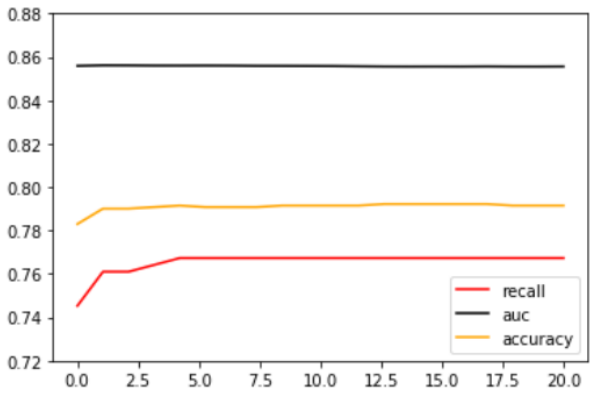
```
###======【TIME WARNING：10mins】======###
import matplotlib.pyplot as plt
C_range = np.linspace(0.01,20,20)

recallall = []
aucall = []
scoreall = []
for C in C_range:
    times = time()
    clf = SVC(kernel = "linear",C=C
              ,class_weight = "balanced").fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest,Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
    recallall.append(recall)
    aucall.append(auc)
    scoreall.append(score)
    print("under C %f, testing accuracy is %f,recall is %f', auc is %f" %
(C,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))

print(max(aucall),C_range[aucall.index(max(aucall))])
plt.figure()
plt.plot(C_range,recallall,c="red",label="recall")
plt.plot(C_range,aucall,c="black",label="auc")
plt.plot(C_range,scoreall,c="orange",label="accuracy")
```

```
plt.legend(loc=4)
plt.ylim([0.72,0.88])
plt.show()
```

这段代码运行大致需要10分钟时间，因此我给大家我展现一下我运行出来的结果：

```
under C 0.010000, testing accuracy is 0.782979,recall is 0.745283', auc is 0.855853
00:00:417883
under C 1.062105, testing accuracy is 0.790071,recall is 0.761006', auc is 0.856100
00:02:293915
under C 2.114211, testing accuracy is 0.790071,recall is 0.761006', auc is 0.856069
00:03:960449
under C 3.166316, testing accuracy is 0.790780,recall is 0.764151', auc is 0.855985
00:05:543221
under C 4.218421, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855977
00:06:899512
under C 5.270526, testing accuracy is 0.790780,recall is 0.767296', auc is 0.855994
00:08:731726
under C 6.322632, testing accuracy is 0.790780,recall is 0.767296', auc is 0.855956
00:10:778214
under C 7.374737, testing accuracy is 0.790780,recall is 0.767296', auc is 0.855861
00:11:335667
under C 8.426842, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855841
00:11:911168
under C 9.478947, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855804
00:15:529542
under C 10.531053, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855766
00:15:778903
under C 11.583158, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855657
00:17:577069
under C 12.635263, testing accuracy is 0.792199,recall is 0.767296', auc is 0.855559
00:20:599942
under C 13.687368, testing accuracy is 0.792199,recall is 0.767296', auc is 0.855536
00:20:332629
under C 14.739474, testing accuracy is 0.792199,recall is 0.767296', auc is 0.855553
00:21:535444
under C 15.791579, testing accuracy is 0.792199,recall is 0.767296', auc is 0.855536
00:22:800108
under C 16.843684, testing accuracy is 0.792199,recall is 0.767296', auc is 0.855591
00:26:582925
under C 17.895789, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855530
00:26:708624
under C 18.947895, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855522
00:27:101574
under C 20.000000, testing accuracy is 0.791489,recall is 0.767296', auc is 0.855562
00:29:954981
```

```
0.8561003985532287 1.0621052631578947
```



可以观察到几个现象：

首先，我们注意到，随着C值逐渐增大，模型的运行速度变得越来越慢。对于SVM这个本来运行就不快的模型来说，巨大的C值会是一个比较危险的消耗。所以正常来说，我们应该设定一个较小的C值范围来进行调整。

其次，C很小的时候，模型的各项指标都很低，但当C到1以上之后，模型的表现开始逐渐稳定，在C逐渐变大之后，模型的效果并没有显著地提高。可以认为我们设定的C值范围太大了，然而再继续增大或者缩小C值的范围，AUC面积也只能够在0.86上下进行变化了，调节C值不能够让模型的任何指标实现质变。

我们把目前为止最佳的C值带入模型，看看我们的准确率，Recall的具体值：

```
times = time()
clf = SVC(kernel = "linear",C=1.0621052631578947
          ,class_weight = "balanced"
         ).fit(Xtrain, Ytrain)
result = clf.predict(Xtest)
score = clf.score(Xtest,Ytest)
recall = recall_score(Ytest, result)
auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
print("testing accuracy %f,recall is %f', auc is %f" % (score,recall,auc))
print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

可以看到，这种情况下模型的准确率，Recall和AUC都没有太差，但是也没有太好，这也许就是模型平衡后的一种结果。现在，光是调整支持向量机本身的参数，已经不能够满足我们的需求了，要想让AUC面积更进一步，我们需要绘制ROC曲线，查看我们是否可以通过调整阈值来对这个模型进行改进。

```
from sklearn.metrics import roc_curve as ROC
import matplotlib.pyplot as plt

FPR, Recall, thresholds = ROC(Ytest,clf.decision_function(Xtest),pos_label=1)

area = roc_auc_score(Ytest,clf.decision_function(Xtest))

plt.figure()
plt.plot(FPR, Recall, color='red',
         label='ROC curve (area = %0.2f)' % area)
plt.plot([0, 1], [0, 1], color='black', linestyle='--')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('Recall')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

以此模型作为基础，我们来求解最佳阈值：

```
maxindex = (Recall - FPR).tolist().index(max(Recall - FPR))
thresholds[maxindex]
```

基于我们选出的最佳阈值，我们来认为确定y_predict，并确定在这个阈值下的recall和准确度的值：

```
from sklearn.metrics import accuracy_score as AC

times = time()
clf = SVC(kernel = "linear",C=1.0621052631578947
          ,class_weight = "balanced"
         ).fit(Xtrain, Ytrain)

prob = pd.DataFrame(clf.decision_function(Xtest))

prob.loc[prob.iloc[:,0] >= thresholds[maxindex],"y_pred"]=1
prob.loc[prob.iloc[:,0] < thresholds[maxindex],"y_pred"]=0

#检查模型本身的准确度
score = AC(Ytest,prob.loc[:,"y_pred"].values)
recall = recall_score(Ytest, prob.loc[:,"y_pred"])
print("testing accuracy %f,recall is %f" % (score,recall))
print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

反而还不如我们不调整时的效果好。可见，如果我们追求平衡，那SVC本身的结果就已经非常接近最优结果了。调节阈值，调节参数C和调节class_weight都不一定有效果。但整体来看，我们的模型不是一个糟糕的模型，但这个结果如果提交到kaggle参加比赛是绝对不足够的。如果大家感兴趣，还可以更加深入地探索模型，或者换别的方法来处理特征，以达到AUC面积0.9以上，或是准确度或recall都提升到90%以上。