Python Tutorial 6

April 4, 2020

This tutorial is for Dr. Xin Tong's DSO 530 class at the University of Southern California in Spring 2020. It contains two parts: the first part is to perform CV by classification error and AUC and the second part is about HW2.

1 Cross-validation by Classification Error and AUC

In this part, we demonstrate k-Fold cross-validation using classification error and AUC with an *auto* classification example. This dataset has been considered in *Python Tutorial 5*.

[1]:
```python
import numpy as np
import pandas as pd

Auto = pd.read_csv('auto.csv')
Auto.head()
```

[1]:
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0  18.0          8         307.0         130    3504          12.0    70
1  15.0          8         350.0         165    3693          11.5    70
2  18.0          8         318.0         150    3436          11.0    70
3  16.0          8         304.0         150    3433          12.0    70
4  17.0          8         302.0         140    3449          10.5    70

   origin                       name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
```

displacement represents a vehicle's engine displacement. First, we transform it into a binary variable *displacement_binary* in two steps: 1) we calculate the mean of *displacement*; 2) we compare each value of *displacement* with the mean of *displacement*. If it is smaller than the mean, we label it as *small*. Otherwise, we label it as *big*. Then we use *displacement_binary* as the responses to do the classification.

[7]:
```python
small_index = Auto["displacement"] <= np.mean(Auto["displacement"])
Auto.loc[small_index,"displacement_binary"] = 'small'
Auto.loc[~small_index,"displacement_binary"] = 'big'
```

Note that we still need to add a column name *displacement_big* to represent *displacement_binary* and make it numeric if we want to use *smf.logit* to do logistic regression.

[8]:
```
Auto["displacement_big"] = np.where(Auto["displacement_binary"] == 'big', 1, 0)
```

[9]:
```
Auto
```

[9]:
```
       mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0     18.0          8         307.0         130    3504          12.0    70
1     15.0          8         350.0         165    3693          11.5    70
2     18.0          8         318.0         150    3436          11.0    70
3     16.0          8         304.0         150    3433          12.0    70
4     17.0          8         302.0         140    3449          10.5    70
..     …          …           …        …      …           …    …
387   27.0          4         140.0          86    2790          15.6    82
388   44.0          4          97.0          52    2130          24.6    82
389   32.0          4         135.0          84    2295          11.6    82
390   28.0          4         120.0          79    2625          18.6    82
391   31.0          4         119.0          82    2720          19.4    82

     origin                       name displacement_binary  displacement_big
0         1  chevrolet chevelle malibu                 big                 1
1         1          buick skylark 320                 big                 1
2         1         plymouth satellite                 big                 1
3         1              amc rebel sst                 big                 1
4         1                ford torino                 big                 1
..      …                        …                 …                 …
387       1            ford mustang gl               small                 0
388       2                 vw pickup               small                 0
389       1              dodge rampage               small                 0
390       1                ford ranger               small                 0
391       1                 chevy s-10               small                 0

[392 rows x 11 columns]
```

We use 10-fold CV to compare two logistic regression models that use different predictors.

First, we use *mpg* and *horsepower* as predictor variables and use *displacement_big* as the response variable.

[10]:
```
from sklearn.model_selection import KFold
kfolds = KFold(n_splits = 10, shuffle = True, random_state = 1)## a random␣
↪state is set for reproducibility purpose
```

[11]:
```
print(kfolds)
```

```
KFold(n_splits=10, random_state=1, shuffle=True)
```

To show the details while implementing *kfolds.split*, in the following execution, we print out the *train_index* and *test_index* of the first loop for you to understand it.

[12]:
```
for train_index, test_index in kfolds.split(Auto):
    print("trian_index:{}\n\ntest_index;{}".format(train_index, test_index))
    break
```

```
trian_index:[  0   1   2   3   7   8   9  10  11  12  13  14  15  16  17  19  20
21
  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57
  58  59  60  61  62  63  64  65  66  68  69  70  71  72  73  74  75  76
  77  79  82  83  84  85  86  87  88  89  90  91  93  94  95  96  97  98
  99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
 117 118 121 122 123 124 125 126 127 128 129 130 131 133 134 135 136 137
 138 139 140 141 142 143 144 145 147 148 149 150 151 152 153 154 155 156
 157 158 159 160 163 164 166 168 169 170 171 172 173 174 175 176 177 178
 179 180 181 182 183 184 186 187 188 189 190 191 192 193 194 195 196 198
 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 215 216 217
 219 220 221 222 223 225 226 227 229 230 231 233 234 235 237 238 239 240
 241 242 243 244 245 246 247 248 249 251 252 253 254 255 256 257 258 259
 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 279
 280 281 282 284 285 286 287 288 289 291 292 293 294 295 296 297 298 299
 300 301 302 303 304 305 306 308 309 310 311 312 313 314 315 316 317 318
 319 320 321 322 323 325 326 327 328 329 330 331 332 334 335 336 337 339
 340 341 342 343 344 345 346 347 348 349 350 352 353 354 355 356 357 358
 359 360 361 362 363 364 365 367 368 369 370 371 372 373 374 375 376 378
 380 381 382 384 386 387 388 389 390 391]

test_index;[  4   5   6  18  67  78  80  81  92 119 120 132 146 161 162 165 167
185
 197 214 218 224 228 232 236 250 260 278 283 290 307 324 333 338 351 366
 377 379 383 385]
```

[13]:
```
cv_classification_errors_1 = []
cv_auc_1 = []
```

[14]:
```
import statsmodels.formula.api as smf
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

for train_index, test_index in kfolds.split(Auto):
    # train the logistic model
    result = smf.logit('displacement_big ~ mpg + horsepower', data=Auto, subset␣
↪= train_index).fit()

    # select the test set according to test_index produced by kfolds.split
    X_test = Auto.loc[test_index,["mpg","horsepower"]]
    y_test = Auto.loc[test_index,"displacement_big"]
```

```
    # compute the probabilities of test data
    result_prob = result.predict(X_test)
    # select 0.5 as the threshold
    result_pred = (result_prob > 0.5)
    # compute the classification error
    classification_error = np.mean(result_pred != y_test)
    # add the computed classification error to "cv_classification_errors_1" to
 ↪store the result
    cv_classification_errors_1.append(classification_error)

    # calculate the auc
    fpr,tpr,threshold = roc_curve(y_test, result_prob)
    roc_auc = auc(fpr,tpr)
    # add the computed auc to "cv_auc_1" to store the result
    cv_auc_1.append(roc_auc)
```

```
Optimization terminated successfully.
         Current function value: 0.231405
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.194249
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.225203
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.212232
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.224298
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.227459
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.232208
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.229884
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.235412
         Iterations 8
Optimization terminated successfully.
         Current function value: 0.227329
         Iterations 9
```

Note that the outputs above are the default output of *smf.logit().fit()* mentioned in *Python Tutorial 3*.

[15]:
```
print("classification errors using 10-fold CV: {}\n".
 ↪format(cv_classification_errors_1))
print("mean of classification errors using 10-fold CV: {}\n".format(np.
 ↪mean(cv_classification_errors_1)))
```

```
classification errors using 10-fold CV: [0.05, 0.175, 0.1794871794871795,
0.15384615384615385, 0.10256410256410256, 0.02564102564102564,
0.10256410256410256, 0.07692307692307693, 0.02564102564102564,
0.07692307692307693]

mean of classification errors using 10-fold CV: 0.09685897435897436
```

[16]:
```
print("auc using 10-fold CV: {}\n".format(cv_auc_1))
print("mean of auc using 10-fold CV: {}\n".format(np.mean(cv_auc_1)))
```

```
auc using 10-fold CV: [0.9824999999999999, 0.9090909090909092,
0.9682539682539684, 0.9293478260869565, 0.9708994708994708, 0.9786096256684492,
0.9881656804733728, 0.9833333333333333, 0.9973262032085561, 0.98]

mean of auc using 10-fold CV: 0.9687527017015019
```

Then, we use *weight* and *acceleration* as predictor variables and use *displacement_big* as the response variable to do the logistic regression.

[17]:
```
kfolds
```

[17]:
```
KFold(n_splits=10, random_state=1, shuffle=True)
```

[18]:
```
cv_classification_errors_2 = []
cv_auc_2 = []
```

[19]:
```
import statsmodels.formula.api as smf
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

for train_index, test_index in kfolds.split(Auto):
    # train the logistic model
    result = smf.logit('displacement_big ~ weight + acceleration', data=Auto,␣
 ↪subset = train_index).fit()

    # select the test set according to test_index produced by kfolds.split
    X_test = Auto.loc[test_index,["weight","acceleration"]]
    y_test = Auto.loc[test_index,"displacement_big"]
```

```
    # compute the probabilities of test data
    result_prob = result.predict(X_test)
    # select 0.5 as the threshold
    result_pred = (result_prob > 0.5)
    # compute the classification error
    classification_error = np.mean(result_pred != y_test)
    # add the computed classification error to "cv_classification_errors_1" to␣
↪store the result
    cv_classification_errors_2.append(classification_error)

    # calculate the auc
    fpr,tpr,threshold = roc_curve(y_test, result_prob)
    roc_auc = auc(fpr,tpr)
    # add the computed auc to "cv_auc_1" to store the result
    cv_auc_2.append(roc_auc)
```

```
Optimization terminated successfully.
         Current function value: 0.171989
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.151944
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.168547
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.166613
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.167117
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.160206
         Iterations 10
Optimization terminated successfully.
         Current function value: 0.155913
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.173752
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.167781
         Iterations 9
Optimization terminated successfully.
         Current function value: 0.156656
         Iterations 9
```

[20]:
```
print("classification errors using 10-fold CV: {}\n".
 ↪format(cv_classification_errors_2))
print("mean of classification errors using 10-fold CV: {}".format(np.
 ↪mean(cv_classification_errors_2)))
```

```
classification errors using 10-fold CV: [0.05, 0.175, 0.07692307692307693,
0.07692307692307693, 0.07692307692307693, 0.10256410256410256,
0.10256410256410256, 0.0, 0.05128205128205128, 0.07692307692307693]

mean of classification errors using 10-fold CV: 0.07891025641025642
```

[21]:
```
print("auc using 10-fold CV: {}\n".format(cv_auc_2))
print("mean of auc using 10-fold CV: {}".format(np.mean(cv_auc_2)))
```

```
auc using 10-fold CV: [0.9974999999999999, 0.9595959595959597,
0.992063492063492, 0.9891304347826088, 0.9894179894179895, 0.9759358288770053,
0.9733727810650887, 1.0, 0.9919786096256684, 0.9828571428571429]

mean of auc using 10-fold CV: 0.9851852238284954
```

Now we can compare the results of the above two models.

[22]:
```
print("predictor varible: mpg, horsepower; response variable: displacement_big")
print("mean of classification errors using 10-fold CV: {}".format(np.
 ↪mean(cv_classification_errors_1)))
print("mean of auc using 10-fold CV: {}\n".format(np.mean(cv_auc_1)))

print("predictor varible: weight, acceleration; response variable:␣
 ↪displacement_big")
print("mean of classification errors using 10-fold CV: {}".format(np.
 ↪mean(cv_classification_errors_2)))
print("mean of auc using 10-fold CV: {}".format(np.mean(cv_auc_2)))
```

```
predictor varible: mpg, horsepower; response variable: displacement_big
mean of classification errors using 10-fold CV: 0.09685897435897436
mean of auc using 10-fold CV: 0.9687527017015019

predictor varible: weight, acceleration; response variable: displacement_big
mean of classification errors using 10-fold CV: 0.07891025641025642
mean of auc using 10-fold CV: 0.9851852238284954
```

With both cross-validation criteria, the model with *weight* and *acceleration* as predictors is the better model.

2 About Question 2 in HW2

2.1 Random Seed and Default Value

One student tried the following two blocks of code but wonder why the samples differ.

[24]:
```
np.random.seed(2)
color = ['red','black','blue','yellow']
for i in range(3):
    print(f'Sample {i+1}: {np.random.choice(a = color, size = 5, replace =␣
 ↪True, p = [0.25, 0.25, 0.25, 0.25])}')
```

```
Sample 1: ['black' 'red' 'blue' 'black' 'black']
Sample 2: ['black' 'red' 'blue' 'black' 'black']
Sample 3: ['blue' 'blue' 'red' 'blue' 'red']
```

[25]:
```
np.random.seed(2)
color = ['red','black','blue','yellow']
for i in range(3):
    print(f'Sample {i+1}: {np.random.choice(a = color, size = 5, replace =␣
 ↪True)}')
```

```
Sample 1: ['red' 'yellow' 'black' 'red' 'blue']
Sample 2: ['yellow' 'blue' 'yellow' 'red' 'yellow']
Sample 3: ['blue' 'black' 'yellow' 'yellow' 'black']
```

As the default option to assign equal probability to each element in the set "color", the above two blocks seem to be doing the same thing but produce different outcomes.

This has to do with how numpy handles the default values. They should be statistically identical, but it is not guaranteed that their implementation is the same. The default option may have a different implementation. And this phenomenon is not unique to this *np.random.choice* function.

2.2 Python Keyword Arguments

When we call a function that includes some specified values for its parameters, these values get assigned to the arguments according to their positions if we skip the key words.

We take Question 2 in HW2 as an example. *np.random.choice* has its default keywords' order: *numpy.random.choice(a, size=None, replace=True, p=None)*.

A standard method to call the *np.random.choice* function is using the following code.

[26]:
```
np.random.seed(2)
print(np.random.choice(a = color, size = 5, replace = True, p = [0.25, 0.25, 0.
 ↪25, 0.25]))
```

```
['black' 'red' 'blue' 'black' 'black']
```

But you can also omit the keywords to call the function like this:

[27]:
```
np.random.seed(2)
print(np.random.choice(color, 5, True, [0.25, 0.25, 0.25, 0.25]))
```

```
['black' 'red' 'blue' 'black' 'black']
```

These values get assigned to the arguments according to their positions.

The value *color* gets assigned to the argument *a* and similarly *5* to *size*, *True* to *replace*, *[0.25, 0.25, 0.25, 0.25]* to *p*.

The following code appears in some answers to the quesion 2 of HW2.

[28]:
```
np.random.seed(2)
print(np.random.choice(color, 5, [0.25, 0.25, 0.25, 0.25]))
```

```
['red' 'yellow' 'black' 'red' 'blue']
```

Actually, the above code is not implementing what we want. *[0.25, 0.25, 0.25, 0.25]* gets assigned to the argument *replace* and gets evaluated as *True*.

By default, an object is considered **True** unless its class defines either a *__bool__*() method that returns *False* or a *__len__*() method that returns zero, when called with the object. Here are most of the built-in objects considered **False**:

1) *constants defined to be false: None and False.*
2) *zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)*
3) *empty sequences and collections: '', (), [], {}, set(), range(0)*

You can validate it to check the following results.

[29]:
```
np.random.seed(2)
print(np.random.choice(color, 5, [0.1, 0.1, 0.1, 0.7]))
```

```
['red' 'yellow' 'black' 'red' 'blue']
```

[30]:
```
np.random.seed(2)
print(np.random.choice(color, 5, True))
```

```
['red' 'yellow' 'black' 'red' 'blue']
```

Therefore, a recommended practice is to keep the key words when you call the functions.

References:

https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.choice.html

https://docs.python.org/3/library/stdtypes.html