Bài 24 - Mất cân bằng dữ liệu (imbalanced dataset)

17 Feb 2020 - phamdinhkhanh

Menu

1. Mất cân bằng dữ liệu (imbalanced dataset)

Mất cân bằng dữ liệu là một trong những hiện tượng phổ biến của bài toán phân loại nhị phân (binary classification) như spam email, phát hiện gian lận, dự báo vỡ nợ, chuẩn đoán bệnh lý,…. Trong trường hợp tỷ lệ dữ liệu giữa 2 classes là 50:50 thì được coi là cân bằng. Khi có sự khác biệt trong phân phối giữa 2 classes, chẳng hạn 60:40 thì dữ liệu có hiện tượng mất cân bằng.

Hầu hết các bộ dữ liệu đều khó đạt được trạng thái cân bằng mà luôn có sự khác biệt về tỷ lệ giữa 2 classes. Đối với những trường hợp dữ liệu mất cân bằng nhẹ như tỷ lệ 60:40 thì sẽ không ảnh hưởng đáng kể tới khả năng dự báo của mô hình.

Tuy nhiên nếu hiện tượng **mất cân bằng nghiêm trọng** xảy ra, chẳng hạn như tỷ lệ 90:10 sẽ thường dẫn tới ngộ nhận chất lượng mô hình. Khi đó thước đo đánh giá mô hình là độ chính xác (accuracy) có thể đạt được rất cao mà không cần tới mô hình. Ví dụ, một dự báo ngẫu nhiên đưa ra tất cả đều là nhóm đa số thì độ chính xác đã đạt được là 90%. Do đó không nên lựa chọn độ chính xác làm chỉ số đánh giá mô hình để tránh lạc quan sai lầm về chất lượng.

Trong trường hợp mẫu mất cân bằng nghiêm trọng ta cần phải thay đổi chỉ số đánh giá để đưa ra kết quả hợp lý hơn. Tôi sẽ trình bày các chỉ số thay thế cho độ chính xác ở phần 4.1 của bài viết này.

Ngoài ra, mất cân bằng dữ liệu nghiêm trọng thường dẫn tới dự báo kém chính xác trên nhóm thiểu số. Bởi đa phần kết quả dự báo ra thường thiên về 1 nhóm là nhóm đa số và rất kém trên nhóm thiểu số. Trong khi tầm quan trọng của việc dự báo được chính xác một mẫu thuộc nhóm thiểu số lớn hơn nhiều so với dự báo mẫu thuộc nhóm đa số. Để cải thiện kết quả dự báo chúng ta cần những điều chỉnh thích hợp để mô hình đạt được một độ chính xác cao trên nhóm thiểu số.

Top

Những điều chỉnh cần thiết giúp cải thiện hiệu năng dự báo của mô hình trong trường hợp xảy ra mất cân bằng nghiêm trọng là gì? Có những phương pháp nào để đối phó với hiện tượng mất cân bằng mẫu nghiêm trọng? Bài viết này tôi sẽ cung cấp một góc nhìn toàn cảnh về các giải pháp chung để giải quyết vấn đề này.

2. Tập dữ liệu

Để thuận lợi cho việc minh họa giải pháp, tôi sẽ xây dựng một mô hình trên bộ dữ liệu gian lận thẻ tín dụng (https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients) và thực hiện các đánh giá đo lường mức độ cải thiện mô hình trước và sau khi áp dụng các điều chỉnh.

Thông tin về bộ dữ liệu:

Đây là bộ dữ liệu về thẻ hành vi gian lận trong 6 tháng đầu năm 2005 tại một ngân hàng thuộc Đài Loan. Bộ dữ liệu bao gồm 30000 các hợp đồng thuộc cả 2 nhóm là vỡ nợ và không vỡ nợ. Mẫu xảy ra hiện tượng mất cân bằng nghiêm trọng vì tỷ lệ `bình thường:vỡ nợ` là `23364:6636`. Chúng ta sẽ cùng xem những giải pháp đưa ra sẽ cải thiện kết quả của mô hình như thế nào.

Thông tin trường:

Trong bộ dữ liệu này chúng ta sẽ dự báo hành vi vỡ nợ của khách hàng trong tháng tới. Biến mục tiêu là default_payment_next_month (Yes = 1, No = 0).

Đầu vào của mô hình là 23 biến còn lại có ý nghĩa như bên dưới:

- ID: Mã số xác định hồ sơ vay. Mỗi một ID ứng với một quan sát duy nhất.
- LIMIT_BAL: Số dư tín dụng bao gồm cả cá nhân người vay và những người phụ thuộc trong gia đình. Đơn vị NT dolar.
- SEX: Giới tính (1 = Nam, 2 = Nữ).
- EDUCATION: Trình độ giáo dục (1 = tốt nghiệp trung học, 2 = đại học, 3 = trung học thông, 4 = khác).
- MARRIAGE: Trạng thái hôn nhân (1 = đã kết hôn, 2 = độc thân, 3 = khác)
- AGE: Độ tuổi.
- PAY_0 - PAY_6: Lịch sử trả nợ trong quá khứ theo tuần tự của tháng. PAY_6 là tháng xa nhất và PAY_0 là tháng gần nhất. Chỉ số lường cho repayment status được chia thành các hạng: -1 = Trả nợ đúng hạn; 1 = trả nợ chậm 1 tháng; 2= trả nợ chậm 2 tháng; …; 9 = trả nợ chậm 9 tháng.
- BILL_AMT1 - BILL_AMT6: Tổng giá trị của bill. BILL_AMT1 là giá trị bill trong tháng gần nhất, tuần tự như thế cho đến giá trị bill trong tháng sau cùng là BILL_AMT6.
- PAY_AMT1 - PAY_AMT6: Số tiền của tháng trước đã thanh toán. PAY_AMT1 là tháng gần nhất cho đến PAY_AMT6 là tháng xa nhất.

3. Phân chia tập train/val/dev/test

Tiếp theo để huấn luyện, lựa chọn và kiểm tra kết quả của mô hình chúng ta sẽ phân chia một cách ngẫu nhiên, không trùng lặp bộ dữ liệu thành các tập train/val/dev/test. Các bộ dữ liệu này có ý nghĩa và vai trò như sau:

- tập train: Dựa trên các biến input và target của tập train, ta sẽ huấn luyện mô hình phân loại vỡ nợ. Mô hình thu được sẽ được đánh giá ở những tập dữ liệu độc lập khác như tập val, dev, và tập test.
- tập val: Đây là tập dữ liệu có các trường tương tự như tập train. Chúng ta chỉ sử dụng tập này để kiểm tra kết quả dự báo của mô hình mà không đưa vào huấn luyện mô hình. Thông qua đánh giá trên tập val, các hiện tượng `overfitting` hoặc `underfitting` nghiêm trọng sẽ được phát hiện và tiến hành hiệu chỉnh.

Top

- tập dev: Đây là tập dữ liệu có các trường cũng tương tự như tập train và val nhưng được dùng để đánh giá việc lựa chọn các siêu tham số (hyper parameters) cho các mô hình huấn luyện từ tập train.
- tập test: Đây cũng là tập dữ liệu có các trường giống train, val, dev và được coi như những quan sát mới hoàn toàn. Tập test nên có phân phối giống nhất với dữ liệu thực tế mà người dùng sẽ tạo ra để đánh giá khả năng áp dụng mô hình vào thực tiễn.

Các tập dữ liệu sẽ được lựa chọn ngẫu nhiên và không trùng lặp. Trong đó tập train có tỷ lệ kích thước giữa `bình thường: vỡ nợ` là `10000:500` , và val/dev/test đều là là `2000:100` để đảm bảo rằng tỷ lệ giữa 2 classes trên các tập train/val/dev/test là cân bằng và mẫu xảy ra hiện tượng mất cân bằng nghiêm trọng.

Đọc dữ liệu:

```
import os
import pandas as pd

dataset = pd.read_csv('default_of_credit_card_clients.csv', header = 0,
                      encoding='utf-8', engine='python')
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
ID                          30000 non-null int64
LIMIT_BAL                   30000 non-null int64
SEX                         30000 non-null int64
EDUCATION                   30000 non-null int64
MARRIAGE                    30000 non-null int64
AGE                         30000 non-null int64
PAY_0                       30000 non-null int64
PAY_2                       30000 non-null int64
PAY_3                       30000 non-null int64
PAY_4                       30000 non-null int64
PAY_5                       30000 non-null int64
PAY_6                       30000 non-null int64
BILL_AMT1                   30000 non-null int64
BILL_AMT2                   30000 non-null int64
BILL_AMT3                   30000 non-null int64
BILL_AMT4                   30000 non-null int64
BILL_AMT5                   30000 non-null int64
BILL_AMT6                   30000 non-null int64
PAY_AMT1                    30000 non-null int64
PAY_AMT2                    30000 non-null int64
PAY_AMT3                    30000 non-null int64
PAY_AMT4                    30000 non-null int64
PAY_AMT5                    30000 non-null int64
PAY_AMT6                    30000 non-null int64
default_payment_next_month  30000 non-null int64
dtypes: int64(25)
memory usage: 5.7 MB
```

Phân chia train/val/dev/test:

Top

```
import numpy as np

model_features = list(set(dataset.columns).difference({"ID", "default
target = ["default_payment_next_month"]
X = dataset[model_features]
y = dataset[target]

id_pos = np.where(y.values.reshape(-1) == 1)[0]
id_neg = np.where(y.values.reshape(-1) == 0)[0]

np.random.shuffle(id_pos)
np.random.shuffle(id_neg)

# Tập train:
id_train_neg = id_neg[:10000]
id_train_pos = id_pos[:500]
id_train = np.concatenate((id_train_neg, id_train_pos), axis = 0)

# Tập val:
id_val_neg = id_neg[10000:12000]
id_val_pos = id_pos[500:600]
id_val = np.concatenate((id_val_neg, id_val_pos), axis = 0)

# Tập dev:
id_dev_neg = id_neg[12000:14000]
id_dev_pos = id_pos[600:700]
id_dev = np.concatenate((id_dev_neg, id_dev_pos), axis = 0)

# Tập test:
id_test_neg = id_neg[14000:16000]
id_test_pos = id_pos[700:800]
id_test = np.concatenate((id_test_neg, id_test_pos), axis = 0)

# khởi tạo dataset
data_train = dataset.iloc[id_train]
data_val = dataset.iloc[id_val]
data_dev = dataset.iloc[id_dev]
data_test = dataset.iloc[id_test]

print('data train shape: ', data_train.shape)
print('data val shape: ', data_val.shape)
print('data dev shape: ', data_dev.shape)
print('data test shape: ', data_test.shape)
```

```
data train shape:  (10500, 25)
data val shape:  (2100, 25)
data dev shape:  (2100, 25)
data test shape:  (2100, 25)
```

4. Các phương pháp giải quyết dữ liệu mất cân bằng

Top

4.1. Thay đổi metric:

Như đã giải thích ở mục đầu tiên, khi hiện tượng mất cân bằng dữ liệu nghiêm trọng xảy ra thì việc sử dụng độ chính xác làm thước đo đánh giá mô hình thường không hiệu quả bởi hầu hết chúng đều đạt độ chính xác rất cao. Một mô hình ngẫu nhiên dự báo toàn bộ là nhãn thuộc nhóm đa số cũng sẽ mang lại kết quả gần bằng `100%`. Khi đó ta có thể cân nhắc tới một số metrics thay thế như `precision, recall, f1-score, gini,...`. Các chỉ số này sẽ không quá lớn để dẫn tới ngộ nhận độ chính xác, đồng thời chúng tập trung hơn vào việc đánh giá độ chính xác trên nhóm thiểu số, nhóm mà chúng ta muốn dự báo chính xác hơn so với nhóm đa số.

		Actual		
		Positive	**Negative**	
Predicted	**Positive**	True Positive (TP)	False Positive (FP) Type I error	Precision = TP/(TP+FP)
	Negative	False Negative (FN) Type II error	True Negative (TN)	
		Recall = TP/(TP+FN)	False positive rate (FPR) = FP/(FP+TN)	

Hình 1: Bảng cross table mô tả kết quả thống kê chéo giữa nhãn dự báo và ground truth. Ở đây Positive tương ứng với nhãn `1` (vỡ nợ) và Negative tương ứng với nhãn `0` (thông thường).

Từ bảng cross table ta dễ dàng hình dung được ý nghĩa của các chỉ số đó là:

- Precision: Mức độ dự báo chính xác trong những trường hợp được dự báo là Positive.

$$precision = \frac{TP}{TP + FP}$$

- Recall: Mức độ dự báo chuẩn xác những trường hợp là Positive trong những trường hợp thực tế là Positive.

$$Recall = \frac{TP}{TP + FN}$$

- F1-Score: Trung bình điều hòa giữa Precision và Recall. Đây là chỉ số thay thế lý tưởng cho accuracy khi mô hình có tỷ lệ mất cân bằng mẫu cao.

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

- Kappa-Score: Là chỉ số đo lường mức độ liên kết tin cậy (inter-rater reliability) cho các categories.
- Gini: Đo lường sự bất bình đẳng trong phân phối giữa Positive và Negative được dự báo từ mô hình.
- AUC: Biểu diễn mối quan hệ giữa độ nhạy (sensitivity) và độ đặc hiệu (specificity). Đánh giá khả năng phân loại good và bad được dự báo từ mô hình.

Một mô hình có các chỉ số trên đều cao thì mô hình có chất lượng dự báo càng tốt.

Trong bài này tôi sẽ sử dụng 2 chỉ số `auc` và `f1 score` là 2 thước đo chính đánh giá mô hình.

Top

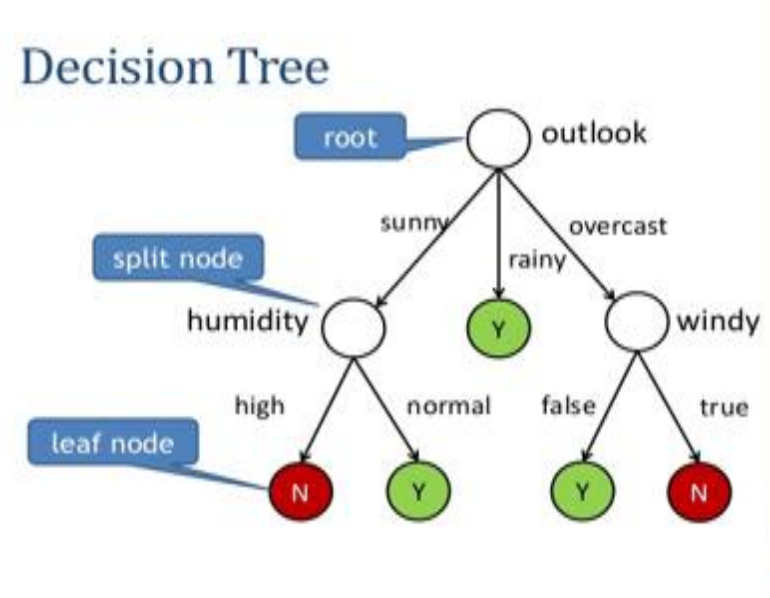
4.2. Xây dựng mô hình

4.2.1. Thuật toán Random forest:

Bên dưới chúng ta sẽ sử dụng thuật toán random forest để huấn luyện mô hình.

Decision Tree

Trước tiên để hiểu về thuật toán Random Forest ta cần hiểu khái niệm về cây quyết định (decision tree).



Hình 2: Sơ đồ cây quyết đinh. Các node là những hình tròn trắng. Các mũi tên liên kết các node với nhau được gọi là nhánh. Một cây quyết định sẽ xuất phát bắt đầu từ root node, sau đó rẽ nhánh tới các split node và trả ra kết quả phân phối xác suất cho quan sát tại leaf node. Mối quan hệ giữa 2 node A và B bất kì có thể là parent-children (cha-con) nếu node A rẽ nhánh trực tiếp sang node B, sibling (anh chị em) nếu node A và B có chung node cha. Rotine là một phương án đường đi liên kết các node thuộc các cấp khác nhau bằng các nhánh.

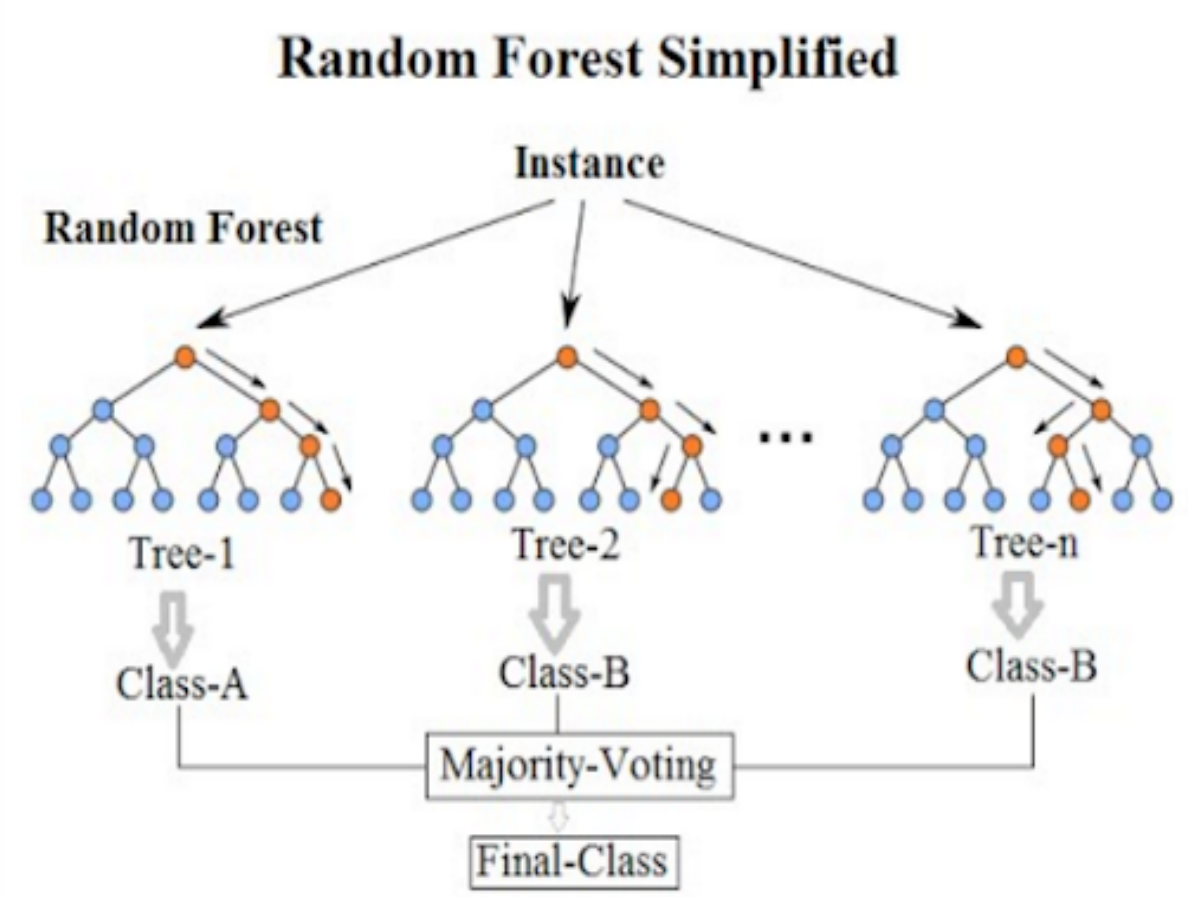
Thuật toán decision tree sẽ xây dựng một cây quyết định ngẫu nhiên dựa trên các node và nhánh. Node bắt đầu của cây quyết định là root node. Từ root node, mô hình sẽ xây dựng một câu hỏi lựa chọn. Tập các phương án có thể là toàn bộ các nhóm của biến category hoặc 2 lựa chọn YES/NO được sinh ra từ biến liên tục. Mỗi một nhánh sẽ tương ứng với một phương án `sunny` (nắng), `rainy` (mưa) hoặc `overcast` (u ám) của root node và kết nối đến một internal node.

Tại các internal node, mô hình tiếp tục rẽ nhánh tới những internal node ở tầng thấp hơn tương ứng với các biến khác. Thứ tự các biến được lựa chọn là ngẫu nhiên. Quá trình rẽ nhánh được thực hiện tiếp tục cho đến khi mô hình đi đến node cuối cùng là leaf node. Tại node này không có nhánh nào được rẽ thêm và trả ra kết quả dự báo của cây quyết định.

Nhãn của kết quả dự báo phụ thuộc vào phân phối xác suất các classes được tính toán theo đường đi từ root node đến leaf node.

Random Forest

Top



Hình 3: Kiến trúc mô hình random forest. Mô hình là một tập hợp của nhiều cây quyết định. Mỗi một cây quyết định sẽ trả ra một kết quả dự báo. Quyết định cuối cùng về nhãn của quan sát sẽ dựa trên nguyên tắc bầu cử đa số (Majority-Voting) trên toàn bộ các cây quyết định con. Ngoài ra mô hình cũng được chạy trên rất nhiều các sub-sample. Nếu một quan sát xuất hiện tại nhiều sub-sample thì sẽ thực hiện bầu cử đa số trên tất cả các cây quyết định của toàn bộ các sub-sample.

Random Forest là thuật toán thuộc lớp mô hình kết hợp (ensemble model). Kết quả của thuật toán dựa trên bầu cư đa số từ nhiều cây quyết định. Do đó mô hình có độ tin cậy cao hơn và độ chính xác tốt hơn so với những mô hình phân loại tuyến tính đơn giản như logistic hoặc linear regression.

Bên cạnh Random Forest thì Gradient Boosting và AdaBoost cũng là các mô hình thuộc lớp mô hình kết hợp thường được áp dụng và mang lại hiệu quả cao tại nhiều cuộc thi.

Tham số của Random Forest:

Sẽ có 3 kịch bản siêu tham số được lựa chọn. Dựa trên kiểm nghiệm từ tập dev, chúng ta quyết định lựa chọn bộ siêu tham số phù hợp nhất.

Để tunning mô hình random forest chúng ta dựa trên một số tham số chính:

- **n_estimators**: Số lượng các trees trên một cây quyết định.
- **max_depth**: Độ sâu lớn nhất của một cây quyết định.
- **min_samples_split**: Số lượng mẫu tối thiểu cần thiết để phân chia một internal node. Nếu kích thước mẫu ở một internal node nhỏ hơn ngưỡng thì ta sẽ không rẽ nhánh internal node.
- **max_features**: Số lượng các features được xem xét khi tìm kiếm phương án phân chia tốt nhất. Mặc định là toàn bộ các features đầu vào.

Top

- **class_weight**: Trọng số tương ứng với mỗi class. Mặc định là None, khi đó các class sẽ có mức độ quan trọng như nhau. Nếu lựa chọn `balance` các class sẽ được đánh trọng số tỷ lệ nghịch với tỷ trọng mẫu của chúng.
- **min_impurity_split**: Ngưỡng để dừng sớm (early stopping) quá trình phát triển của cây quyết định. Nó sẽ tiếp tục phân chia nếu độ vẩn đục (impurity) lớn hơn ngưỡng threshold, trái lại thì nó là node leaf.

Chúng ta sẽ có một số kịch bản mô hình như sau:

- Mô hình 1:

n_estimators=100, max_depth=5, min_samples_split=200, class_weight=None, max_features=10

- Mô hình 2:

n_estimators=500, max_depth=10, min_samples_split=400, max_features="auto", class_weight="balanced", max_features=20

- Mô hình 3:

n_estimators=800, max_depth=10, min_samples_split=200, max_features="sqrt", class_weight="balanced"

Top

```
import numpy as np
from sklearn.calibration import calibration_curve, CalibratedClassifi
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, f1_score
import matplotlib.pyplot as plt

model1 = RandomForestClassifier(n_estimators=100,
                                max_depth=5,
                                min_samples_split=200,
                                class_weight=None,
                                max_features=10)

model2 = RandomForestClassifier(n_estimators=500,
                                max_depth=10,
                                min_samples_split=400,
                                random_state=12,
                                class_weight="balanced",
                                max_features="auto")

model3 = RandomForestClassifier(n_estimators=800,
                                max_depth=10,
                                min_samples_split=200,
                                random_state=12,
                                class_weight="balanced",
                                max_features="sqrt")

def _tunning_model(model , X_train, y_train, X_dev, y_dev):
  model.fit(X_train, y_train)
  model_predictions = model.predict_proba(X_dev)
  model_roc_score = roc_auc_score(y_dev,
                                  model_predictions[:,1])
  return model, model_roc_score

model1, model1_roc_score = _tunning_model(model1,
                                          data_train[model_features],
                                          data_dev[model_features], da
print('model 1 ROC score on dev dataset: ', model1_roc_score)


model2, model2_roc_score = _tunning_model(model2,
                                          data_train[model_features],
                                          data_dev[model_features], da
print('model 2 ROC score on dev dataset: ', model2_roc_score)

model3, model3_roc_score = _tunning_model(model3,
                                          data_train[model_features],
                                          data_dev[model_features], da
print('model 3 ROC score on dev dataset: ', model3_roc_score)
```

```
model 1 ROC score on dev dataset:  0.7648699999999999
model 2 ROC score on dev dataset:  0.7701
model 3 ROC score on dev dataset:  0.768555
```

Top

Như vậy bằng kiểm tra trên tập dev set cho thấy mô hình 2 sẽ có kết quả tốt nhất. Do đó ta sẽ coi mô hình 2 làm mô hình baseline và các siêu tham số của nó sẽ được giữ để khởi tạo các mô hình về sau.

4.3. Under sampling

Under sampling là việc ta giảm số lượng các quan sát của nhóm đa số để nó trở nên cân bằng với số quan sát của nhóm thiểu số. Ưu điểm của under sampling là làm cân bằng mẫu một cách nhanh chóng, dễ dàng tiến hành thực hiện mà không cần đến thuật toán giả lập mẫu.

Tuy nhiên nhược điểm của nó là kích thước mẫu sẽ bị giảm đáng kể. Gỉa sử nhóm thiểu số có kích thước là 500, như vậy để tạo ra sự cân bằng mẫu giữa nhóm đa số và thiểu số sẽ cần giảm kích thước mẫu của nhóm đa số từ 10000 về 500. Tổng kích thước tập huấn luyện sau under sampling là 1000 và chiếm gần 1/10 kích thước tập huấn luyện ban đầu. Tập huấn luyện mới khá nhỏ, không đại diện cho phân phối của toàn bộ tập dữ liệu và thường dễ dẫn tới hiện tượng overfitting.

Do đó trong một số phương án, chúng ta có thể không nhất thiết lựa chọn sao cho tỷ lệ mẫu giữa `nhóm đa số: nhóm thiểu số` là `50:50` mà có thể giảm dần xuống về `80:20`, `70:30` hoặc `60:40` và tìm ra phương án nào mang lại hiệu quả dự báo tốt nhất trên tập kiểm tra.

Bên dưới ta sẽ xây dựng mô hình trên 2 tỷ lệ mẫu `80:20` và `70:30` và đánh giá mô hình trên tập test.

```
# Phân chia mẫu ngẫu nhiên theo tỷ lệ 80:20 bằng cách giữ lại 2000 mẫ
np.random.shuffle(id_train_neg)
id_train_neg_80_20 = id_train_neg[:2000]
id_train_80_20 = np.concatenate((id_train_neg_80_20, id_train_pos), a

# Phân chia mẫu ngẫu nhiên theo tỷ lệ 70:30 bằng cách giữ lại 1166 mẫ
np.random.shuffle(id_train_neg)
id_train_neg_70_30 = id_train_neg[:1166]

id_train_70_30 = np.concatenate((id_train_neg_70_30, id_train_pos), a

# khởi tạo dataset
data_train_80_20 = dataset.iloc[id_train_80_20]
data_train_70_30 = dataset.iloc[id_train_70_30]
```

Top

```
# Huấn luyện mô hình trên mẫu tỷ lệ 80:20
model2_unsam_80_20 = RandomForestClassifier(n_estimators=500,
                                            max_depth=10,
                                            min_samples_split=400,
                                            random_state=12,
                                            class_weight="balanced",
                                            max_features="auto")

model2_unsam_80_20.fit(data_train_80_20[model_features], data_train_8
model_predictions = model2_unsam_80_20.predict_proba(data_test[model_
model_pred_label = model2_unsam_80_20.predict(data_test[model_feature
model_roc_score = roc_auc_score(data_test['default_payment_next_month
model_f1_score = f1_score(data_test['default_payment_next_month'], mo
print('model2_unsam_80_20 roc score on test: ', model_roc_score)
print('model2_unsam_80_20 f1 score on test: ', model_f1_score)

# Huấn luyện mô hình trên mẫu tỷ lệ 70:30
model2_unsam_70_30 = RandomForestClassifier(n_estimators=500,
                                            max_depth=10,
                                            min_samples_split=400,
                                            random_state=12,
                                            class_weight="balanced",
                                            max_features="auto")

model2_unsam_70_30.fit(data_train_70_30[model_features], data_train_7
model_predictions = model2_unsam_70_30.predict_proba(data_test[model_
model_pred_label = model2_unsam_70_30.predict(data_test[model_feature
model_roc_score = roc_auc_score(data_test['default_payment_next_month
model_f1_score = f1_score(data_test['default_payment_next_month'], mo
print('model2_unsam_70_30 roc score on test: ', model_roc_score)
print('model2_unsam_70_30 f1 score on test: ', model_f1_score)
```

```
model2_unsam_80_20 roc score on test:  0.7702749999999998
model2_unsam_80_20 f1 score on test:  0.19999999999999998
model2_unsam_70_30 roc score on test:  0.7649575000000001
model2_unsam_70_30 f1 score on test:  0.1949778434268833
```

Dự báo mô hình trên tập test và tính toán các chỉ số `auc` và `f1 score` .

```
model_predictions = model2.predict_proba(data_test[model_features])
model_pred_label = model2.predict(data_test[model_features])
model_roc_score = roc_auc_score(data_test['default_payment_next_month'
model_f1_score = f1_score(data_test['default_payment_next_month'], mod
print('random forest roc score on test: ', model_roc_score)
print('random forest f1 score on test: ', model_f1_score)
```

```
random forest roc score on test:  0.782075
random forest f1 score on test:  0.245398773006135
```

So sanh kết quả cho ta thấy khi thực hiện under sampling đã không giúp cải thiện được nhiều kết quả dự báo trên tập test ở cả 2 chỉ số `auc` và `f1 score` . Nguyên nhân có thể là do việc thực hiện over sampling đã dẫn tới overfitting. Chúng ta cần thử nghiệm với nhiều tỷ lệ mẫu khác nhau

Top

hơn để tìm ra tỷ lệ phù hợp. Phần này xin dành cho bạn đọc.

4.4. Over sampling

Over sampling là các phương pháp giúp giải quyết hiện tượng mất cân bằng mẫu bằng cách gia tăng kích thước mẫu thuộc nhóm thiểu số bằng các kĩ thuật khác nhau. Có 2 phương pháp chính để thực hiện over sampling đó là:

- Lựa chọn mẫu có tái lập.
- Mô phỏng mẫu mới dựa trên tổng hợp của các mẫu cũ.

Chúng ta cùng tìm hiểu cụ thể các phương pháp này ở phần bên dưới.

4.4.1. Naive random over-sampling

Naive random Over sampling là phương pháp tái chọn mẫu dựa trên giả thuyết ngây ngô là dữ liệu mẫu giả lập mới sẽ giống dữ liệu sẵn có. Do đó ta sẽ cân bằng mẫu bằng cách lựa chọn ngẫu nhiên có lặp lại các quan sát thuộc nhóm thiểu số.

Trong ví dụ này tôi sẽ lựa chọn Naive random over-sampling sao cho tỷ lệ mẫu giữa 2 nhóm là cân bằng. Giữ nguyên các mẫu thuộc nhóm đa số và tăng kích thước mẫu thuộc nhóm thiểu số sao cho bằng với nhóm đa số. Sau đó huấn luyện model trên tập mẫu đã được over sampling và kiểm tra kết quả trên tập test. So sánh với kết quả từ mô hình baseline để đánh giá mức độ cải thiện.

```
from collections import Counter

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import make_classification
from sklearn.svm import LinearSVC

from imblearn.pipeline import make_pipeline
from imblearn.over_sampling import RandomOverSampler
from imblearn.base import BaseSampler
```

```
model_smote = RandomForestClassifier(n_estimators=500,
                                     max_depth=10,
                                     min_samples_split=400,
                                     random_state=12,
                                     class_weight="balanced",
                                     max_features="auto")

pipe = make_pipeline(RandomOverSampler(sampling_strategy=1, random_sta
pipe.fit(data_train[model_features], data_train['default_payment_next_
```

Top

```
Pipeline(memory=None,
         steps=[('randomoversampler',
                 RandomOverSampler(random_state=0, ratio=None,
                                   return_indices=False, sampling_str
                ('randomforestclassifier',
                 RandomForestClassifier(bootstrap=True, ccp_alpha=0.0
                                        class_weight='balanced',
                                        criterion='gini', max_depth=1
                                        max_features='auto',
                                        max_leaf_nodes=None, max_samp
                                        min_impurity_decrease=0.0,
                                        min_impurity_split=None,
                                        min_samples_leaf=1,
                                        min_samples_split=400,
                                        min_weight_fraction_leaf=0.0,
                                        n_estimators=500, n_jobs=None
                                        oob_score=False, random_state
                                        verbose=0, warm_start=False))
         verbose=False)
```

Dự báo và kiểm định trên test

```
model_smote_predictions = pipe.predict_proba(data_test[model_features]
model_smote_pred_label = pipe.predict(data_test[model_features])
model_smote_roc_score = roc_auc_score(data_test['default_payment_next_m
model_smote_f1_score = f1_score(data_test['default_payment_next_month'
print('random forest roc score on test: ', model_smote_roc_score)
print('random forest f1 score on test: ', model_smote_f1_score)
```

```
random forest roc score on test:  0.7825150000000001
random forest f1 score on test:  0.2565217391304348
```

```
model_predictions = model2.predict_proba(data_test[model_features])
model_pred_label = model2.predict(data_test[model_features])
model_roc_score = roc_auc_score(data_test['default_payment_next_month'
model_f1_score = f1_score(data_test['default_payment_next_month'], mod
print('random forest roc score on test: ', model_roc_score)
print('random forest f1 score on test: ', model_f1_score)
```

```
random forest roc score on test:  0.782075
random forest f1 score on test:  0.245398773006135
```

Như vậy ta thấy rằng khi sử dụng phương pháp random over sampling mẫu sẽ giúp cải thiện được đáng kể kết quả dự báo. Chỉ số `auc` tăng lên từ: `0.7820` lên `0.7825` trên tập test và chỉ số `f1 score` tăng từ `0.2454` lên `0.2565`.

4.4.2. SMOTE & ADASYN

Top

SMOTE (Synthetic Minority Over-sampling) và ADASYN (Adaptive synthetic sampling) là các phương pháp sinh mẫu nhằm gia tăng kích thước mẫu của nhóm thiểu số trong trường hợp xảy ra mất cân bằng mẫu. Để gia tăng kích thước mẫu, với mỗi một mẫu thuộc nhóm thiểu số ta sẽ lựa chọn ra k mẫu láng giềng gần nhất với nó và sau đó thực hiện tổ hợp tuyến tính để tạo ra mẫu giả lập. Phương pháp để lựa chọn ra các láng giềng của một quan sát có thể dựa trên thuật toán `kNN` hoặc `SVM` .

Chi tiết các thuật toán này tôi sẽ không trình bày tại đây. Các bạn có thể tham khảo tại Synthetic Minority Over-sampling (https://arxiv.org/pdf/1106.1813.pdf) và Adasyn adaptive synthetic (https://towardsdatascience.com/adasyn-adaptive-synthetic-sampling-method-for-imbalanced-data-602a3673ba16).

Bên dưới ta sẽ cùng áp dụng phương pháp over-sampling từ SMOTE và ADASYN và kiểm tra kết quả dự báo của mô hình được huấn luyện trên mẫu over-sampling.

```
from imblearn.pipeline import make_pipeline
from imblearn.over_sampling import ADASYN
from imblearn.over_sampling import (SMOTE, BorderlineSMOTE, SVMSMOTE,
from imblearn.over_sampling import RandomOverSampler
from imblearn.base import BaseSampler

rf_clf = RandomForestClassifier(n_estimators=500,
                                max_depth=10,
                                min_samples_split=400,
                                random_state=12,
                                class_weight="balanced",
                                max_features="auto")

smotes = {0 : 'SMOTE',
          1 : 'BorderlineSMOTE',
          2 : 'SVMSMOTE',
          3 : 'ADASYN'}


for i, sampler in enumerate((SMOTE(sampling_strategy = 1, random_stat
                 BorderlineSMOTE(sampling_strategy = 1, random_state=0
                 SVMSMOTE(sampling_strategy = 1, random_state=0),
                 ADASYN(sampling_strategy = 1, random_state=0))):
    pipe_line = make_pipeline(sampler, rf_clf)
    pipe_line.fit(data_train[model_features], data_train['default_pay
    rf_predictions = pipe_line.predict_proba(data_test[model_features
    rf_pred_label = pipe_line.predict(data_test[model_features])
    rf_roc_score = roc_auc_score(data_test['default_payment_next_mont
    rf_f1_score = f1_score(data_test['default_payment_next_month'], r
    print('------------------------------------------------')
    print('SMOTE method: ', smotes[i])
    print('random forest roc score on test: ', rf_roc_score)
    print('random forest f1 score on test: ', rf_f1_score)
```

Top

```
----------------------------------------------
SMOTE method:  SMOTE
random forest roc score on test:  0.71945
random forest f1 score on test:  0.2255965292841649


----------------------------------------------
SMOTE method:  BorderlineSMOTE
random forest roc score on test:  0.767965
random forest f1 score on test:  0.2519280205655527


----------------------------------------------
SMOTE method:  SVMSMOTE
random forest roc score on test:  0.776185
random forest f1 score on test:  0.28387096774193554


----------------------------------------------
SMOTE method:  ADASYN
random forest roc score on test:  0.71786
random forest f1 score on test:  0.21940928270042193
```

Như vậy ta thấy 2 phương pháp SMOTE là `SVMSMOTE` và `BorderlineSMOTE` đã giúp tăng chỉ số `f1 score` so với mô hình baseline. Trong đó phương pháp `SVMSMOTE` có mức độ cải thiện f1 score là gần 4% từ `0.2454` lên `0.2839`. Trong khi f1 score thay đổi không đáng kể. Đây là cải thiện rất tốt đối với một mô hình mà hầu hết kết quả của `f1` chỉ xoay quanh khoảng từ `0.24-0.25`.

4.5. Thu thập thêm quan sát

Thông thường với các mô hình mà số lượng quan sát trong nhóm thiểu số quá nhỏ sẽ không thể đại diện cho toàn bộ các nguyên nhân dẫn đến quan sát rơi vào nhóm thiểu số. Để mô hình học được bao quát hơn các khả năng, chúng ta cần gia tăng kích thước mẫu thiểu bằng cách thu thập thêm các quan sát thực tế thuộc nhóm thiểu số.

Ví dụ, giả sử ta thu thập thêm 500 quan sát thuộc nhóm thiểu số vào tập train. Các quan sát này phải thỏa mãn điều kiện chưa từng xuất hiện ở các mẫu của tập test để tạo tính khách quan khi đánh giá mô hình. Trên nguyên tắc, mẫu đã được đưa vào huấn luyện mô hình sẽ không được sử dụng để kiểm tra mô hình.

Như vậy ta sẽ có một tập dữ liệu mới là `data_train_add` trong code bên dưới bao gồm 10000 mẫu bình thường và 1000 mẫu vỡ nợ. Cùng xây dựng mô hình và kiểm tra trên tập test.

Top

```
# Tập train:
id_train_neg = id_neg[:10000]
id_train_pos = id_pos[:500]
# Lấy ngẫu nhiên 500 quan sát thuộc nhóm thiểu không trùng với test
id_train_pos_add = id_pos[800:1300]
id_train_add = np.concatenate((id_train_neg, id_train_pos, id_train_p
# khởi tạo dataset
data_train_add = dataset.iloc[id_train_add]
print('data train shape: ', data_train_add.shape)
```

```
data train shape:  (11000, 25)
```

```
model_add = RandomForestClassifier(n_estimators=800,
                                   max_depth=10,
                                   min_samples_split=200,
                                   random_state=12,
                                   class_weight="balanced",
                                   max_features="sqrt")

model_add.fit(data_train_add[model_features], data_train_add['default_
model_add_predictions = model_add.predict_proba(data_test[model_featu
model_add_pred_label = model_add.predict(data_test[model_features])
model_add_roc_score = roc_auc_score(data_test['default_payment_next_m
model_add_f1_score = f1_score(data_test['default_payment_next_month']
print('random forest roc score on test: ', model_add_roc_score)
print('random forest f1 score on test: ', model_add_f1_score)
```

```
random forest roc score on test:  0.79414
random forest f1 score on test:  0.2629310344827586
```

Như vậy ta nhận thấy thu thập thêm dữ liệu cho nhóm thiểu số cũng là một phương án cải thiện khả năng dự báo so với mô hình baseline, chỉ số `auc` và `f1 score` đều tăng.

4.6. Thu thập thêm biến

Mô hình có kết quả kém có thể là do đang thiếu những biến quan trọng có ảnh hưởng lớn tới xác định hành vi của nhóm thiểu số. Chẳng hạn đối với bài toán dự báo khả năng vỡ nợ, chúng ta có thể thu thập thêm dữ liệu về lịch sử nợ xấu trên toàn bộ hệ thống ngân hàng, mức thu nhập, đã có nhà chưa, đã có xe chưa, số người phụ thuộc,…. Đây là những biến sẽ cung cấp thêm những thông tin hữu ích để nhận diện tốt hơn những trường hợp có khả năng vỡ nợ.

Hiểu biết lĩnh vực (knownledge domain) rất quan trọng đối với data scientist trước khi xây dựng mô hình. Có nhiều biến đầu vào quan trọng không dễ dàng nhận biết nếu data scientist không có hiểu biết về lĩnh vực đang phân loại. Để bổ sung thêm biến, thu thập ý kiến chuyên gia là một biện pháp quan trọng để tạo ra bộ dữ liệu chất lượng cho huấn luyện mô hình. Các chuyên gia là những người có kinh nghiệm và hiểu biết chuyên sâu về đặc tính của các nhóm. Do đó họ sẽ đưa ra nhiều rules nhận diện giúp ích cho phân loại.

Top

4.7. Phạt mô hình

Việc dự báo sai một quan sát thuộc mẫu đa số sẽ ít nghiêm trọng hơn so với dự báo sai một quan sát thuộc mẫu thiểu số. Xuất phát từ ý tưởng đó chúng ta sẽ phạt nặng hơn đối với sai số dự báo thuộc nhóm thiểu bằng cách gán cho nó một trọng số lớn hơn trong công thức của hàm loss function. Chẳng hạn như bên dưới trong argument `class_weight` chúng ta sẽ gán cho các trọng số của class thiểu số nhãn `1` là giá trị 0.9 và class đa số nhãn `0` là 0.1. Khi đó kết quả mô hình thu được trên tập test sẽ là:

```
model_pen = RandomForestClassifier(n_estimators=500,
                                   max_depth=10,
                                   min_samples_split=400,
                                   random_state=12,
                                   class_weight={0: 0.1,
                                                 1: 0.9},
                                   max_features="auto")

model_pen.fit(data_train[model_features], data_train['default_payment
model_pen_predictions = model_pen.predict_proba(data_test[model_featu
model_pen_pred_label = model_pen.predict(data_test[model_features])
model_pen_roc_score = roc_auc_score(data_test['default_payment_next_m
model_pen_f1_score = f1_score(data_test['default_payment_next_month']
print('random forest roc score on test: ', model_pen_roc_score)
print('random forest f1 score on test: ', model_pen_f1_score)
```

```
random forest roc score on test:  0.784845
random forest f1 score on test:  0.3092105263157895
```

Ta thấy cả `auc` và `f1 score` đều cao hơn so với baseline model. Trong đó `f1` đã tăng gần 6% từ `0.2454` lên tới `0.3`. Đây là kết quả khả quan nhất trong các mô hình từ trước đến nay.

4.8. Thử nghiệm nhiều phương pháp khác nhau.

Quá trình thực nghiệm cho thấy có thuật toán cho kết quả tốt trên các bộ dữ liệu mất cân bằng nghiêm trọng nhưng kém hiệu quả trên các bộ dữ liệu không bị mất cân bằng và ngược lại. Do đó điều chúng ta không nên tin tưởng vào một thuật toán mà phải mở rộng và thử nghiệm mô hình trên nhiều thuật toán khác nhau.

Hãy cùng thử nghiệm huấn luyện một loạt các thuật toán trên tập train và kiểm tra mức độ dự báo chính xác trên tập test.

Logistic Regression

Top

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(C = 0.0001)

def _train_and_test(model, algo):
  model.fit(data_train[model_features], data_train['default_payment_n
  predictions = model.predict_proba(data_test[model_features])
  pred_label = model.predict(data_test[model_features])
  roc_score = roc_auc_score(data_test['default_payment_next_month'],
  model_f1_score = f1_score(data_test['default_payment_next_month'],
  print('{} roc score on test: {}'.format(algo , roc_score))
  print('{} f1 score on test: {}'.format(algo, model_f1_score))
  return model

log_reg = _train_and_test(log_reg, algo = 'Logistic')
```

```
Logistic roc score on test: 0.6628700000000001
Logistic f1 score on test: 0.0


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as sho
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options
    https://scikit-learn.org/stable/modules/linear_model.html#logisti
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

kNN

```
from sklearn.neighbors import KNeighborsClassifier

knn_classifier = KNeighborsClassifier(n_neighbors = 5,
                                       weights = 'distance',
                                       algorithm = 'kd_tree',
                                       metric = 'minkowski'
                                       )

knn_classifier = _train_and_test(knn_classifier, algo = 'kNN')
```

```
kNN roc score on test: 0.5585575
kNN f1 score on test: 0.0
```

SVM

Top

```
from sklearn.svm import LinearSVC
from sklearn.calibration import CalibratedClassifierCV

svm_classifier = LinearSVC(penalty='l2',
                           loss='squared_hinge',
                           tol=0.0001,
                           C=0.9,
                           dual=False,
                           class_weight='balanced',
                           max_iter=1000
                          )
svm_classifier = CalibratedClassifierCV(svm_classifier)
svm_classifier = _train_and_test(svm_classifier, algo = 'SVM')
```

```
SVM roc score on test: 0.71934
SVM f1 score on test: 0.03773584905660377
```

MLP

```
from tensorflow.keras.layers import Dense, Input, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

# # concate nate all layers
# encode_els = concatenate(encode_els)
inputlayer = Input(shape=(23,))
hidden1 = Dense(units = 128, kernel_initializer = 'normal', activatio
droplayer1 = Dropout(0.8)(hidden1)
hidden2 = Dense(64, kernel_initializer = 'normal', activation = 'relu
droplayer2 = Dropout(0.2)(hidden2)
outputlayer = Dense(1, kernel_initializer = 'normal', activation = 's
mlp_classifier = Model(inputs = inputlayer, outputs = [outputlayer])
mlp_classifier.summary()
```

Top

```
WARNING:tensorflow:Large dropout rate: 0.8 (>0.5). In TensorFlow 2.x,
Model: "model_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_14 (InputLayer)        [(None, 23)]              0
_________________________________________________________________
dense_39 (Dense)             (None, 128)               3072
_________________________________________________________________
dropout_26 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_40 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_27 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_41 (Dense)             (None, 1)                 65
=================================================================
Total params: 11,393
Trainable params: 11,393
Non-trainable params: 0
_________________________________________________________________
```

```
optimizer = Adam()
mlp_classifier.compile(optimizer, loss='binary_crossentropy', metrics=
mlp_classifier.fit(data_train[model_features], data_train['default_payı
                   validation_data = (data_val[model_features], data_vı
                   epochs = 10,
                   batch_size = 64)
```

Top

```
Train on 10500 samples, validate on 2100 samples
Epoch 1/10
10500/10500 [==============================] - 1s 90us/sample - loss:
Epoch 2/10
10500/10500 [==============================] - 0s 44us/sample - loss:
Epoch 3/10
10500/10500 [==============================] - 0s 42us/sample - loss:
Epoch 4/10
10500/10500 [==============================] - 0s 43us/sample - loss:
Epoch 5/10
10500/10500 [==============================] - 0s 42us/sample - loss:
Epoch 6/10
10500/10500 [==============================] - 0s 43us/sample - loss:
Epoch 7/10
10500/10500 [==============================] - 0s 42us/sample - loss:
Epoch 8/10
10500/10500 [==============================] - 0s 42us/sample - loss:
Epoch 9/10
10500/10500 [==============================] - 0s 43us/sample - loss:
Epoch 10/10
10500/10500 [==============================] - 0s 44us/sample - loss:



<tensorflow.python.keras.callbacks.History at 0x7fd201605cc0>
```

```
predictions = mlp_classifier.predict(data_test[model_features])
pred_label = [0 if prob <= 0.5 else 1 for prob in predictions]
roc_score = roc_auc_score(data_test['default_payment_next_month'], pre
model_f1_score = f1_score(data_test['default_payment_next_month'], pre
print('{} roc score on test: {}'.format('MLP' , roc_score))
print('{} f1 score on test: {}'.format('MLP', model_f1_score))
```

```
MLP roc score on test: 0.515965
MLP f1 score on test: 0.0
```

Light Gradient Boosting

Top

```
import lightgbm as lgb

lgb_classifier = lgb.LGBMClassifier(n_estimator = 800,
                                    objective = 'binary',
                                    class_weight = 'balanced',
                                    learning_rate = 0.05,
                                    reg_alpha = 0.1,
                                    reg_lambda = 0.1,
                                    subsample = 0.8,
                                    n_job = -1,
                                    random_state = 12
                                   )

lgb_classifier = _train_and_test(lgb_classifier, algo = 'Light Gradie
```

```
Light Gradient Boosting roc score on test: 0.7818150000000001
Light Gradient Boosting f1 score on test: 0.25654450261780104
```

Ta thấy hầu hết các mô hình đều không mang lại kết quả tốt đối với bộ dữ liệu này. Một số mô hình như `kNN, MLP, Logistic` dự báo trên tập test chỉ rơi vào 1 nhóm dẫn tới `f1 score` = 0.

Tuy nhiên, mô hình `Light Gradient Boosting` lại cho kết quả dự báo khá tốt khi cải thiện được cả 2 chỉ số `auc` và `f1` so với baseline model. Thực tế tôi từng xây dựng nhiều mô hình có hiện tượng mất cân bằng dữ liệu nghiêm trọng, và kết quả cho thấy `Light Gradient Boosting` là mô hình có hiệu quả cao trong nhiều bài toán của tôi.

5. Kết luận

Như vậy tôi đã giới thiệu xong một số các phương pháp chính đối phó với hiện tượng mất cân bằng dữ liệu. Trong quá trình xây dựng mô hình, đặc biệt là các mô hình phân loại nhị phân (2 classes) các bạn sẽ thường xuyên gặp lại hiện tượng này. Mất cân bằng dữ liệu sẽ dẫn tới mô hình dự báo kém chính xác và đa phần kết quả dự báo bị thiên về nhãn đa số. Trong trường hợp đó, các thước đo như accuracy cũng không phải là một metric tốt để đánh giá mô hình. Qua bài viết này các bạn sẽ có thêm những phương pháp hữu hiệu để đối phó với các tình huống mất cân bằng dữ liệu. Tùy vào từng bài toán và từng bộ dữ liệu mà data scientist có thể lựa chọn một hoặc kết hợp một vài phương pháp để cải thiện hiệu năng mô hình.

6. Tài liệu

1. imbalanced package doc - python (https://imbalanced-learn.readthedocs.io/en/stable/auto_examples/over-sampling/plot_comparison_over_sampling)
2. probability calibration for imbalanced dataset (https://towardsdatascience.com/probability-calibration-for-imbalanced-dataset-64af3730eaab)
3. best metric measure accuracy classification models (https://www.kdnuggets.com/2016/12/best-metric-measure-accuracy-classification-models.html/2)
4. undersampling and oversampling imbalanced data (https://www.kaggle.com/residentmario/undersampling-and-oversampling-imbalanced-data)

Top

5. dealing with imbalanced data (ttps://www.marcoaltini.com/blog/dealing-with-imbalanced-data-undersampling-oversampling-and-proper-cross-validation)
6. SMOTE: Synthetic Minority Over-sampling Technique - Nitesh V. Chawla vs el. (https://arxiv.org/pdf/1106.1813.pdf)

Top