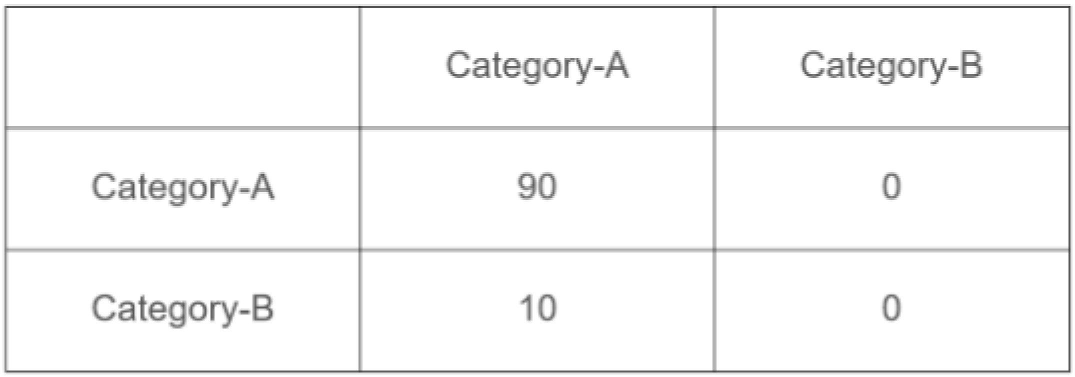
How to deal with imbalanced datasets in Python?

What is an imbalanced dataset?

Imagine, you have two categories in your dataset to predict ('Target variable values) — Category-A and Category-B. When Category-A is higher than Category-B or vice versa, you have a problem of imbalanced dataset.

So how is this a problem?

Imagine in a dataset of 100 rows, Category-A is containing 90 records and Category-B is containing 10 records. You run a machine learning model and end up with 90% accuracy. You were excited until you checked the confusion matrix.



	Category-A	Category-B
Category-A	90	0
Category-B	10	0

Confusion Matrix

Here, Category-B is completely classified as Category-A and the model got away with an accuracy of 90%.

How do we fix this problem?

There are some common and simple ways to deal with this problem.

1. SMOTE
2. NearMiss

SMOTE - (Synthetic Minority Over-sampling Technique)

SMOTE is an over-sampling method. What it does is, it creates synthetic (not duplicate) samples of the minority class. Hence making the minority class equal to the majority class. SMOTE does this by selecting similar records and altering that record one column at a time by a random amount within the difference to the neighboring records.

Let us see how this works in Python:

1. Import the required libraries

```
from imblearn.over_sampling import SMOTE
```

supported libraries:

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, recall_score
```

2. Upload the imbalanced dataset

I'll be using Bank Marketing dataset from UCI. Download "bank.zip" file.

```
# Read the data
bank = pd.read_csv("bank-full.csv", sep = ";", na_values = "unknown")
```

Note: na_values will be filled with "unknown"

```
# Print the column names
print (bank.columns)

# Print top 5 observations
print (bank.head())

# Print shape of the dataset
print (bank.shape)
```

Output:

```
Index(['age', 'job', 'marital', 'education', 'default', 'balance', 'housing',
       'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays',
       'previous', 'poutcome', 'y'],
      dtype='object')
   age           job  marital  education  ... pdays  previous poutcome   y
0   58    management  married   tertiary  ...    -1         0      NaN  no
1   44    technician   single  secondary  ...    -1         0      NaN  no
2   33  entrepreneur  married  secondary  ...    -1         0      NaN  no
3   47   blue-collar  married        NaN  ...    -1         0      NaN  no
4   33           NaN   single        NaN  ...    -1         0      NaN  no

[5 rows x 17 columns]
(45211, 17)
```

The dataset has 45211 observations, and 17 columns.

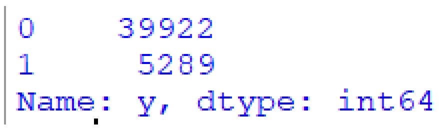
3. Modify the column values as needed (numerical values)

```
# Change the column values
bank["default"] = bank["default"].map({"no":0,"yes":1})
bank["housing"] = bank["housing"].map({"no":0,"yes":1})
bank["loan"] = bank["loan"].map({"no":0,"yes":1})
bank["y"] = bank["y"].map({"no":0,"yes":1})
bank.education = bank.education.map({"primary": 0, "secondary":1,
"tertiary":2})
bank.month = pd.to_datetime(bank.month, format = "%b").dt.month
```

4. Check the value counts on the target column

```
# Target variable value counts
print (bank.y.value_counts())
```

output:



```
0    39922
1     5289
Name: y, dtype: int64
```

5. Drop the observations having null values in their columns and check the value counts again. I also dropped 2 features that are not needed.

```
# Drop the 2 columns
bank.drop(["poutcome", "contact"], axis = 1, inplace = True)

# Total null values / columns
print (bank.isnull().sum())

# Drop null value rows
bank.dropna(inplace = True)

# Target variable value counts
print (bank.y.value_counts())

# Print shape of the dataset
print (bank.shape)
```

output:

```
age              0
job            288
marital          0
education     1857
default          0
balance          0
housing          0
loan             0
day              0
month            0
duration         0
campaign         0
pdays            0
previous         0
y                0
dtype: int64
0    38172
1     5021
Name: y, dtype: int64
(43193, 15)
```

The dataset has now reduced to 43193 observations.

Category – 0	Category – 1
38172	5021
88.38%	11.62%

This clearly shows that the dataset is imbalanced.

6. Introduce dummy variables (encoding) on the categorical features:

```
bank = pd.get_dummies(bank, drop_first = True)
print (bank.columns)
```

output:

```
Index(['age', 'education', 'default', 'balance', 'housing', 'loan', 'day',
       'month', 'duration', 'campaign', 'pdays', 'previous', 'y',
       'job_blue-collar', 'job_entrepreneur', 'job_housemaid',
       'job_management', 'job_retired', 'job_self-employed', 'job_services',
       'job_student', 'job_technician', 'job_unemployed', 'marital_married',
       'marital_single'],
      dtype='object')
```

7. Split the dataset (train and test) and train the model

```
X = bank.drop("y", axis = 1)
y = bank.y

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 1,
stratify=y)
y_train.value_counts()
lr = LogisticRegression()
lr.fit(X_train, y_train)

print ("Value Count 1: ", y_train.value_counts())
print ("Value Count 2: ", y_test.value_counts())
```

Output:

Train dataset: - 32394 records (number of records)

```
Value Count 1:  0    28628
1      3766
Name: y, dtype: int64
```

Category – 0	Category – 1
28628	3766
88.38%	11.62%

Test dataset: - 10799 (number of records)

```
Value Count 2:  0    9544
1     1255
Name: y, dtype: int64
```

Category – 0	Category – 1
9544	1255
88.38%	11.62%

Note: The hyper parameter “stratify = y” splits the records in equal numbers.

8. Test the model

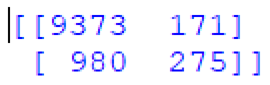
```
y_pred = lr.predict(X_test)

print (accuracy_score(y_test, y_pred))

print (confusion_matrix(y_test, y_pred))
```

```
0.8934160570423187
```

Great 89%! Now let us check the confusion matrix:



```
[[9373   171]
 [ 980   275]]
```

We can clearly see this is a **bad** model. This model is not able to classify clients who have subscription for term deposits. Let's check the recall score:

```
print (recall_score(y_test, y_pred))
```

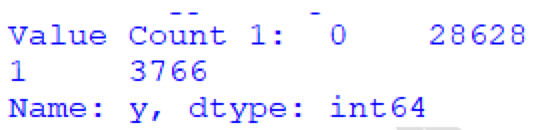
output:

Clearly bad. This is expected, since the one category has more records than the other.

Applying SMOTE:

Before fitting SMOTE, let us check the y_train values:

```
print ("Value Count 1: ", y_train.value_counts())
```



```
Value Count 1:  0    28628
1     3766
Name: y, dtype: int64
```

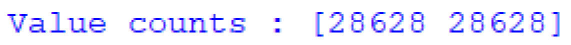
Let us fit SMOTE. You can check out all the parameters from here

```
# Apply SMOTE
smt = SMOTE()
X_train, y_train = smt.fit_sample(X_train, y_train)
```

Now let us check the amount of records in each category:

```
# Count of records
print ("Value counts :", np.bincount(y_train))
```

output:



```
Value counts : [28628 28628]
```

Both categories have equal amount of records. More specifically, the minority class has been increased to the total number of majority class.

Now fitting the classifier again and testing we get an accuracy score of:

```
# Train the model with SMOTEd values
lr = LogisticRegression()
lr.fit(X_train, y_train)

# Predict post SMOTE
y_pred = lr.predict(X_test)

print ('Accuracy score post SMOTE :', accuracy_score(y_test, y_pred), '\n')
```

output:

```
Accuracy score post SMOTE : 0.8032225206037596
```

Whoa! We have reduced the accuracy. But let us check the confusion matrix anyway:

```
print ('Confusion Matrix :', confusion_matrix(y_test, y_pred), '\n')

print ('Recall score post SMOTE :', recall_score(y_test, y_pred), '\n')
```

output:

```
Confusion Matrix : [[7660 1884]
 [ 241 1014]]

Recall score post SMOTE : 0.8079681274900399
```

This is a good model compared to the previous one. Recall is great.

I would go ahead with this model than using the previous model. Now let us check what happens if we use NearMiss.

Applying NearMiss

```
## NearMiss

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 1,
stratify=y)

nr = NearMiss()

X_train, y_train = nr.fit_sample(X_train, y_train)

np.bincount(y_train)

lr = LogisticRegression()
lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)

print ('Confusion Matrix - NearMiss:', confusion_matrix(y_test, y_pred),
'\n')

print ('Accuracy score post NearMiss :', accuracy_score(y_test, y_pred),
'\n')

print ('Recall score post NearMiss :', recall_score(y_test, y_pred), '\n')
```

output:

```
Confusion Matrix - NearMiss: [[5102 4442]
 [ 162 1093]]

Accuracy score post NearMiss : 0.573664228169275

Recall score post NearMiss : 0.8709163346613545
```

This model is better than the first model because it classifies better. But since in this case, SMOTE is giving me a great accuracy and recall, I'll go ahead and use that model!

Full code:

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, recall_score
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import NearMiss

import warnings
warnings.filterwarnings("ignore")

# Read the data
bank = pd.read_csv("bank-full.csv", sep = ";", na_values = "unknown")

# Print the column names
print (bank.columns)

# Print top 5 observations
print (bank.head())

# Print shape of the dataset
print (bank.shape)

# Change the column values
bank["default"] = bank["default"].map({"no":0,"yes":1})
bank["housing"] = bank["housing"].map({"no":0,"yes":1})
bank["loan"] = bank["loan"].map({"no":0,"yes":1})
bank["y"] = bank["y"].map({"no":0,"yes":1})
bank.education = bank.education.map({"primary": 0, "secondary":1,
"tertiary":2})
bank.month = pd.to_datetime(bank.month, format = "%b").dt.month

# Target variable value counts
print (bank.y.value_counts())

# Drop the 2 columns
bank.drop(["poutcome", "contact"], axis = 1, inplace = True)

# Total null values / columns
print (bank.isnull().sum())

# Drop null value rows
bank.dropna(inplace = True)

# Target variable value counts
print (bank.y.value_counts())

# Print shape of the dataset
```

```
print (bank.shape)


bank = pd.get_dummies(bank, drop_first = True)
print (bank.columns)

X = bank.drop("y", axis = 1)
y = bank.y

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 1, 
stratify=y)
y_train.value_counts()
lr = LogisticRegression()
lr.fit(X_train, y_train)

print ("Value Count 1: ", y_train.value_counts())
print ("Value Count 2: ", y_test.value_counts())

y_pred = lr.predict(X_test)

print (confusion_matrix(y_test, y_pred))

print (accuracy_score(y_test, y_pred))

print (recall_score(y_test, y_pred))

print ('\n')

# Apply SMOTE
smt = SMOTE()
X_train, y_train = smt.fit_sample(X_train, y_train)

# Count of records
print ("Value counts :", np.bincount(y_train))

# Train the model with SMOTEd values
lr = LogisticRegression()
lr.fit(X_train, y_train)

# Predict post SMOTE
y_pred = lr.predict(X_test)

print ('Accuracy score post SMOTE :', accuracy_score(y_test, y_pred), '\n')

print ('Confusion Matrix :', confusion_matrix(y_test, y_pred), '\n')

print ('Recall score post SMOTE :', recall_score(y_test, y_pred), '\n')

## NearMiss

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 1, 
stratify=y)
```

```
nr = NearMiss()

X_train, y_train = nr.fit_sample(X_train, y_train)

np.bincount(y_train)

lr = LogisticRegression()
lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)

print ('Confusion Matrix - NearMiss:', confusion_matrix(y_test, y_pred),
'\n')

print ('Accuracy score post NearMiss :', accuracy_score(y_test, y_pred),
'\n')

print ('Recall score post NearMiss :', recall_score(y_test, y_pred), '\n')
```

Reference:
https://medium.com/@saeedAR/smote-and-near-miss-in-python-machine-learning-in-imbalanced-datasets-b7976d9a7a79