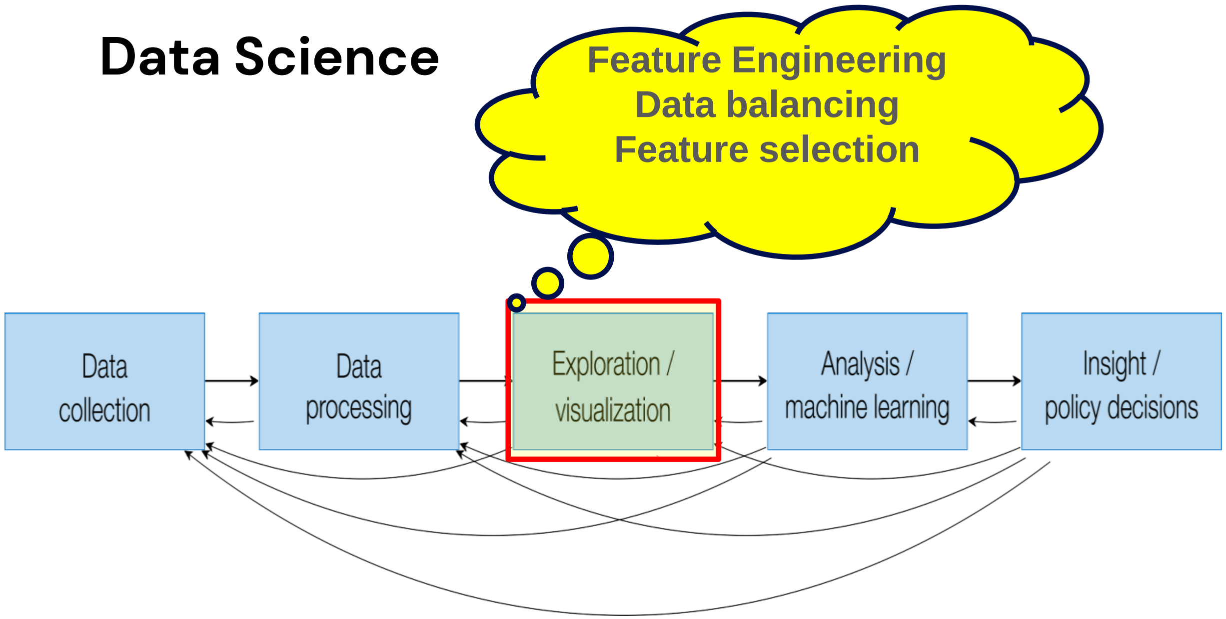
Data Science: Concepts and Practice

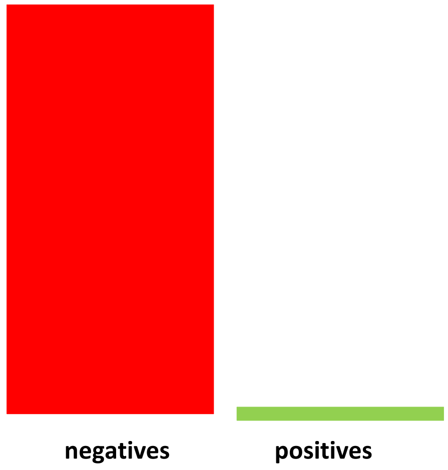
Edited and presented by :
Marwa Al-Hadi

Lecture 6 : Data Balancing

Data Science



- After plotting your class distribution
- you see that you have thousands of negative examples but just a couple of positives.



Classifiers try to reduce the overall error so they can be biased towards the majority class.

Negatives = 998

Positives = 2

By always predicting a negative class the accuracy will be 99.8%

Your dataset is imbalanced!!!

Now What???

The Class Imbalance Problem

- The problem with class imbalances is that standard learners are often biased towards the majority class.

The Class Imbalance Problem

As a result, examples from the overwhelming class are well-classified whereas examples from the minority class tend to be misclassified.

Solutions

Solutions to Imbalanced Learning

- **Data Level** — Sampling methods
- **Algorithmic Level** — Cost-sensitive methods
- **Active Learning Methods** — Kernel and Active Learning methods

Several Common Approaches

- At the data Level: Re-Sampling
  - Oversampling (Random or Directed)
    - Add more examples to minority class
  - Undersampling (Random or Directed)
    - Remove samples from majority class

Several Common Approaches

At the Algorithmic Level:

- Adjusting the *Costs* or *weights* of classes
- Adjusting the decision *threshold* / probabilistic estimate at the tree leaf
- What is threshold???!!!

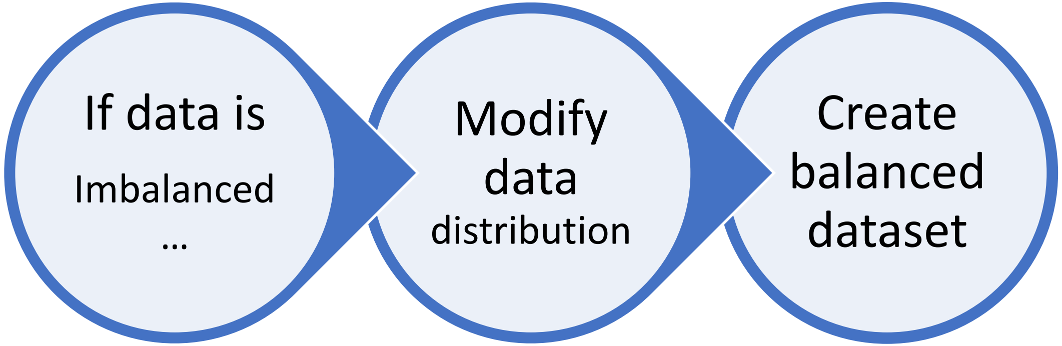
Several Common Approaches

The threshold

- in most models (*e.g., logistic regression, decision trees)*
- is 0.5 , but adjusting this threshold can significantly impact performance, especially for imbalanced datasets.
- *convert a model's probabilistic output into a specific class*
- *how this probability is interpreted*
- **Above the Threshold**: Predict positive class (e.g., $P(y=1) \geq \text{threshold}$).
- **Below the Threshold**: Predict negative class (e.g., $P(y=0)$).

Sampling Methods

Create balance through sampling



A widely adopted technique for dealing with highly unbalanced datasets is called resampling.

Data Level — Sampling methods

Algorithmic Level — Cost-sensitive methods

Active Learning Methods — Kernel and Active Learning methods

Sampling methods

SMOTE

SMOTE: Resampling Approach

- **SMOTE** stands for:

  Synthetic **M**inority **O**versampling **T**echnique

- It is a technique designed by Hall et. al in 2002.

- **SMOTE** is an oversampling method that synthesizes new plausible examples in the minority class.

SMOTE: Resampling Approach

- SMOTE not only increases the size of the training set, but also increases the variety!!
- SMOTE currently yields the best results as far as re-sampling and modifying the probabilistic estimate techniques go (Chawla, 2003).

SMOTE's Informed Oversampling Procedure

For each Minority Sample

I. Find its k-nearest *minority neighbors*

II. Randomly *select* j of these neighbors

III. Randomly *generate synthetic samples* along the lines joining the minority sample and its j selected neighbors

(j depends on the amount of oversampling desired)

```python
import pandas as pd
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from collections import Counter

# Step 3: Read the dataset from a CSV file
# Replace 'your_dataset.csv' with the path to your actual CSV file
df = pd.read_csv('your_dataset.csv')

# Step 4: Separate the features (X) and target variable (y)
# Assuming the target column is named 'target', replace with your actual colum
X = df.drop(columns=['target'])  # Features
y = df['target']  # Target variable

# Step 5: Check the class distribution before SMOTE
print(f"Class distribution before SMOTE: {Counter(y)}")

# Step 6: Apply SMOTE to balance the dataset (on the entire data)
smote = SMOTE(sampling_strategy='auto', random_state=42)  # 'auto' balances al
X_resampled, y_resampled = smote.fit_resample(X, y)

# Step 7: Check the class distribution after SMOTE
print(f"Class distribution after SMOTE: {Counter(y_resampled)}")

# Step 8: Train a model (e.g., RandomForest) on the resampled dataset
model = RandomForestClassifier(random_state=42)
model.fit(X_resampled, y_resampled)

# Step 9: Evaluate the model (use the same data for simplicity, but this is no
y_pred = model.predict(X_resampled)

# Step 10: Print the classification report
print("Classification Report:")
print(classification_report(y_resampled, y_pred))
```

What else instead of SMOTE

- 1. Random Oversampling
- 2. Random Undersampling
- 3. Tomek Links
- 4. NearMiss
- 5. Borderline-SMOTE
- 6. ADASYN (Adaptive Synthetic Sampling)
- 7. Cluster Centroids
- 8. Synthetic Minority Over-sampling Technique for Nominal and Continuous (SMOTE-NC)
- 9. MDO (Modified Distribution Over-sampling)
- 10. Ensemble Learning-Based Methods (Balanced Random Forest, EasyEnsemble)

- **Data Level** — Sampling methods
- **Algorithmic Level** — Cost-sensitive methods
- **Active Learning Methods** — Kernel and Active Learning methods

Cost-Sensitive LR

or **Imbalanced learning** focuses on **how an intelligent system can learn** when it is provided with imbalanced data.

Cost-Sensitive Approach

- **Cost-sensitive learning** is a
- process to minimize the misclassification costs **by**

  incorporating a cost matrix or class weights into the model.
- deal with dataset as it is unbalancing
- costs of prediction errors
- Using of weight to measure

Cost-Sensitive Approach

- **Cost-sensitive learning** is a
- subfield of machine learning or deep learning
- It is a field of study that is **closely related to the field of imbalanced learning**
- that is concerned with classification on datasets with a skewed class distribution.
- As such, adopted for imbalanced classification problems.

Algorithm Steps

Input:

1. Training data: $X = \{x_1, x_2, \ldots, x_n\}$
2. Labels: $Y = \{y_1, y_2, \ldots, y_n\}, y_i \in \{0, 1\}$
3. Cost matrix:

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

- C_{00}: Cost of correctly predicting class 0.
- C_{01}: Cost of predicting class 1 when the true label is class 0.
- C_{10}: Cost of predicting class 0 when the true label is class 1.
- C_{11}: Cost of correctly predicting class 1.

Cost-Sensitive Approach

- For example
- using of Logistic regression as cost sensitive approach, we calculate loss per example using binary cross-entropy:

$$\boxed{H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))}$$

  - Loss = −y log(p(y)) – (1−y) log(1−p(y))
  - where y is the label (1 for class A and 0 for class B)
  - p(y) is the predicted probability of the point being class A.

Cost-Sensitive Approach

if we set class_weight as class_weight = **{0:1,1:20}**, the classifier in the background tries to minimize:

$$\boxed{NewLoss = -20*y \log(p(y)) + 1*(1-y) \log(1-p(y))}$$

- That means *we discipline* our model around *20 times more* when it misclassifies a positive minority example in this case.

Cost-Sensitive approach

- What else …
  - 1. Cost-Sensitive Decision Trees
  - 2. Cost-Sensitive Random Forest
  - 3. Cost-Sensitive Support Vector Machines (SVM)
  - 4. Cost-Sensitive K-Nearest Neighbors (KNN)
  - 6. Cost-Sensitive Neural Networks
  - 7. Cost-Sensitive Gradient Boosting

Assessment Metrics

How to evaluate the performance of imbalanced learning algorithms ?

1. Singular assessment metrics
2. Receiver operating characteristics (ROC) curves
3. Precision-Recall (PR) Curves
4. Cost Curves
5. Assessment Metrics for Multiclass Imbalanced Learning

Assessment Metrics

Singular Assessment Metrics

		True class	
		p	n
Hypothesis output	Y	TP (True Positives)	FP (False Positives)
	N	FN (False Negatives)	TN (True Negatives)
Column counts:		P_C	N_C

$$Accuracy = \frac{TP + TN}{P_C + N_C}$$

$$ErrorRate = 1 - accuracy$$

Assessment Metrics

Singular Assessment Metrics

$$Precision = \frac{TP}{TP + FP},$$

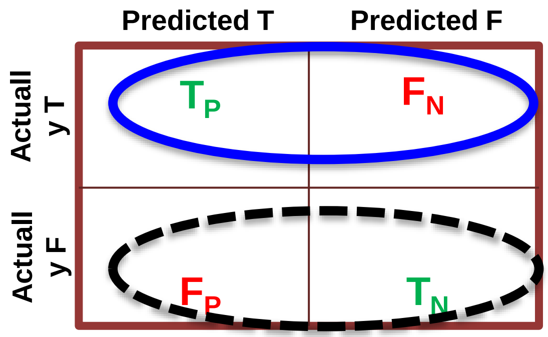
$$Recall = \frac{TP}{TP + FN},$$

- **Precision**: It tells us how correct (precise) our model's positive predictions.
- **Recall (Sensitivity)**: is the ratio of correctly predicted positive classes to all items that are actually positive

	Predicted T	Predicted F
Actually T	T_P	F_N
Actually F	F_P	T_N

Insensitive to data distributions

TPR and TNR



$$TPR = \frac{TP}{Actual\ Positive} = \frac{TP}{TP + FN}$$

$$TNR = \frac{TN}{Actual\ Negative} = \frac{TN}{TN + FP}$$

- True Positive Rate (TPR) is the probability that an actual positive will test positive (Sensitivity/Recall).
- True Negative Rate (TNR) is the probability that an actual negative will test negative (called Specificity).

SKLearn Example

SKLearn Code

- The dataset is about Abalone.
- Abalone, is a species of marine snails.
- There are 4174 instances with **8 features** for each record
  - % of Negative instances: **99.23%**
  - % of Positive instances: **0.77%**

- Our goal is to identify whether an abalone belongs to a specific class. (Positives → 19), (Negative all remaining).
- So, this is a binary classification problem of either positive (class 19) or negative.

- You can download the data from the following link
  - https://github.com/liannewriting/YouTube-videos-public/tree/main/imbalanced-data-machine-learning-abalone19

https://www.youtube.com/watch?v=xFErz6I-FyE&list=PL2L4c5jChmctqiXvOaJA91o0OJhYq1rR9&index=1

SKLearn Code

```
# How to handle Imbalanced Data in machine learning classification
# The slides presented are based on the following Tutorial
# https://www.justintodata.com/imbalanced-data-machine-learning-classification/
# This tutorial will focus on imbalanced data in machine learning for binary
classes,
# but you could extend the concept to multi-class.

import pandas as pd
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import ClusterCentroids
from imblearn.combine import SMOTETomek
from imblearn.under_sampling import TomekLinks

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.utils import compute_class_weight
```

SKLearn Code

```
# Read the dataset
df = pd.read_csv('abalone19.dat')
df.head()
```

	Sex	Length	Diameter	Height	W_weight	S_weight	V_weight	Shell_weight	Class
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	negative
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	negative
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	negative
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	negative
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	negative

SKLearn Code

```
# Find out more about the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4174 entries, 0 to 4173
Data columns (total 9 columns):
 #   Column           Non-Null Count   Dtype
---  ---------        --------------   --------
 0   Sex              4174 non-null    object
 1   Length           4174 non-null    float64
 2   Diameter         4174 non-null    float64
 3   Height           4174 non-null    float64
 4   Whole_weight     4174 non-null    float64
 5   Shucked_weight   4174 non-null    float64
 6   Viscera_weight   4174 non-null    float64
 7   Shell_weight     4174 non-null    float64
 8   Class            4174 non-null    object

dtypes: float64(7), object(2)
memory usage: 293.6+ KB
```

SKLearn Code

```
# Produce some stats on the dataset
df.describe()
```

	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	Class	Sex_I	Sex_M
Count	4174.0	4174.0	4174.0	4174.0	4174.0	4174.0	4174.0	4174.000000	4174.0	4174.0
Mean	0.5240	0.4079	0.139524	0.828771	0.359361	0.180607	0.238853	0.007667	0.321275	0.365597
Std	0.1200	0.0991	0.041818	0.490065	0.221771	0.109574	0.139143	0.087233	0.467022	0.481655
Min	0.0750	0.0550	0.000000	0.002000	0.001000	0.000500	0.001500	0.000000	0.000000	0.000000
25%	0.4500	0.3500	0.115000	0.442125	0.186500	0.093500	0.130000	0.000000	0.000000	0.000000
50%	0.5450	0.4250	0.140000	0.799750	0.336000	0.171000	0.234000	0.000000	0.000000	0.000000
75%	0.6150	0.4800	0.165000	1.153000	0.501875	0.252875	0.328875	0.000000	1.000000	1.000000
Max	0.8150	0.6500	1.130000	2.825500	1.488000	0.760000	1.005000	1.000000	1.000000	1.000000

SKLearn Code

```
# We'll use the most basic machine learning classification algorithm: logistic regression.
# It is better to convert all the categorical columns for logistic regression to dummy variables.
# we'll convert the categorical columns (Sex and Class) within the dataset before modeling.
# Lets look at the category of Sex
# Three Classes: Male, Infant and Female

df['Sex'].value_counts()
```

```
M    1526
I    1341
F    1307
Name: Sex, dtype: int64
```

```
# Lets look at the category of Class
# Two Classes: Negative and Positive
df['Class'].value_counts()
```

```
negative    4142
positive      32
Name: Class, dtype: int64
```

SKLearn Code

```
# Let us convert the Class label into 0 and 1
df['Class'] = df['Class'].map(lambda x: 0 if x == 'negative' else 1)
df
```

	Sex	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	Class
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.1500	0
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.0700	0
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.2100	0
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.1550	0
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.0550	0
...	...	...	...	...	...	...	...	...	...
4169	M	0.560	0.430	0.155	0.8675	0.4000	0.1720	0.2290	0
4170	F	0.565	0.450	0.165	0.8870	0.3700	0.2390	0.2490	0
4171	M	0.590	0.440	0.135	0.9660	0.4390	0.2145	0.2605	0
4172	M	0.600	0.475	0.205	1.1760	0.5255	0.2875	0.3080	0
4173	F	0.625	0.485	0.150	1.0945	0.5310	0.2610	0.2960	0

4174 rows × 9 columns

SKLearn Code

```
# Let us convert the Sex feature into two dummy variables
df = pd.get_dummies(df, columns=['Sex'], drop_first=True)
df
```

	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	Class	Sex_I	Sex_M
0	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.1500	1	0	0
1	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.0700	1	0	0

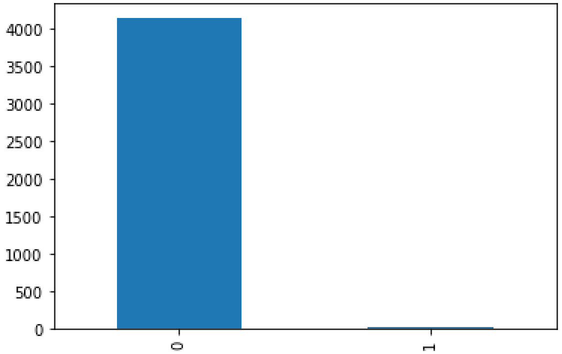
4174 rows × 10 columns

SKLearn Code

```
df['Class'].value_counts(normalize=True)
```

```
0    0.992333
1    0.007667
Name: Class, dtype: float64
```

```
df['Class'].value_counts().plot(kind='bar')
```



SKLearn Code

```
# Splitting Training and Testing sets
# Let’s split the dataset into training (80%) and test sets (20%).
# Use the train_test_split function with stratify argument based on Class categories.
# So that both the training and test datasets will have similar portions of classes as
# the complete dataset.
# This is important for imbalanced data.

df_train, df_test = train_test_split(df, test_size=0.2, stratify=df['Class'],
random_state=888)

features = df_train.drop(columns=['Class']).columns
```

SKLearn Code

```
# Two sets: df_train and df_test.
# We'll use df_train for modeling, and df_test for evaluation.
# Print the different classes (0 and 1) that are present in the Training Set
df_train['Class'].value_counts()
```

```
Training Data
0    3313
1      26
Name: Class, dtype: int64
```

```
# Print the different classes (0 and 1) that are present in the Testing Set
df_test['Class'].value_counts()
```

```
Testing Data
0    829
1      6
Name: Class, dtype: int64
```

SKLearn Code

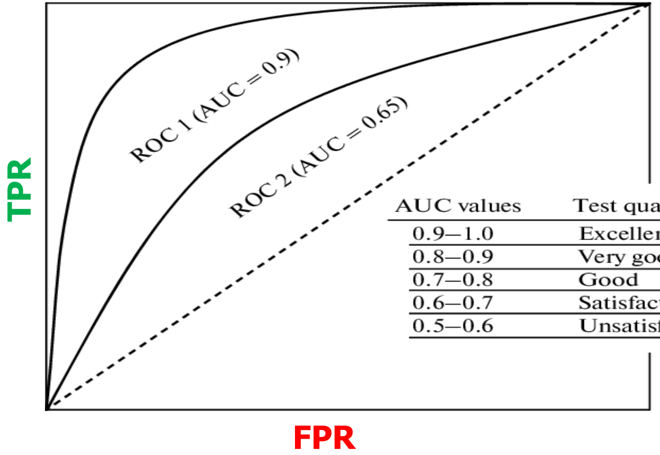
```
# Let us train a Logistic Regression with the unbalanced Data and check the auc
clf = LogisticRegression(random_state=888)

features = df_train.drop(columns=['Class']).columns
clf.fit(df_train[features], df_train['Class'])

y_pred = clf.predict_proba(df_test[features])[:, 1]

print("The AUC score for this model using the original unbalanced data ...")
roc_auc_score(df_test['Class'], y_pred)
```

The AUC score for this model using the original unbalanced data ...
0.683956574185766



AUC values	Test quality
0.9—1.0	Excellent
0.8—0.9	Very good
0.7—0.8	Good
0.6—0.7	Satisfactory
0.5—0.6	Unsatisfactory

SKLearn Code

```
# we could use the library imbalanced-learn to random oversample.

from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE

ros = RandomOverSampler(random_state=888)
X_resampled, y_resampled = ros.fit_resample(df_train[features],
df_train['Class'])
y_resampled.value_counts()
```

```
0    3313
1    3313
Name: Class, dtype: int64
```

SKLearn Code

```
# We can then apply Logistic Regression and calculate the AUC metric.
clf = LogisticRegression(random_state=888)
clf.fit(X_resampled, y_resampled)
y_pred = clf.predict_proba(df_test[features])[:, 1]

print("The AUC score for this model after Random Over Sampling  ...")
roc_auc_score(df_test['Class'], y_pred)
```

The AUC score for this model **after Random Over Sampling** ...
0.838962605548854

SKLearn Code

```
# Random sampling is easy, but the new samples don’t add more information.
# SMOTE improves on that.
# SMOTE oversamples the minority class by creating ‘synthetic’ examples.
# It involves some methods (nearest neighbors), to generate plausible examples.
print("Oversampling using SMOTE ...")
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=888)
X_resampled, y_resampled = smote.fit_resample(df_train[features],
df_train['Class'])

y_resampled.value_counts()
```

```
Oversampling using SMOTE ...
0    3313
1    3313
Name: Class, dtype: int64
```

SKLearn Code

```
# We'll apply logistic regression on the balanced dataset and calculate its AUC.

clf = LogisticRegression(random_state=888)
clf.fit(X_resampled, y_resampled)
y_pred = clf.predict_proba(df_test[features])[:, 1]
print("The AUC score for this model after SMOTE ...")
roc_auc_score(df_test['Class'], y_pred)
```

```
The AUC score for this model after SMOTE
...
0.7913148371531966
```

SKLearn Code

```
# Now we will use Undersampling
# Undersampling, we will downsize majority class to balance with the minority class.
# Simple random undersampling
# We'll begin with simple random undersampling.
rus = RandomUnderSampler(random_state=888)
X_resampled, y_resampled = rus.fit_resample(df_train[features], df_train['Class'])

y_resampled.value_counts()
```

```
0    26
1    26
Name: Class, dtype: int64
```

SKLearn Code

```
# And this produces the same AUC as pandas undersampling, since we use the same
clf = LogisticRegression(random_state=888)
clf.fit(X_resampled, y_resampled)
y_pred = clf.predict_proba(df_test[features])[:, 1]
print("The AUC score for this model after Under Sampling ...")
roc_auc_score(df_test['Class'], y_pred)
```

```
The AUC score for this model after Under Sampling ...
0.6465621230398071
```

SKLearn Code

```
# Weighing classes differently
# We can also balance the classes by weighing the data differently
# We usually consider each observation equally, with a weight value of 1
# But for imbalanced datasets, we can balance the classes by putting more weight
# on the minority classes.
# The below code estimates weights for our imbalanced training dataset.

weights = compute_class_weight('balanced', classes=df_train['Class'].unique(),
y=df_train['Class'])
print("If we want the dataset to be balanced, we need the following weights for
Majority vs Minority ..")
weights
```

```
If we want the dataset to be balanced, we need the following weights for
Majority vs Minority ..

array([ 0.50392394, 64.21153846])
```

SKLearn Code

```
# Let's verify that these weights can indeed balance the dataset.
# Multiply the counts of each class by their respective weights.

print("Performing the following re-wieghting of classes we get ..")
print((df_train['Class'] == 0).sum()*weights[0])
print((df_train['Class'] == 1).sum()*weights[1])
```

```
Performing the following re-wieghting of classes we get ..
1669.5
1669.5000000000002
```

```
# If we sum up the weights of both classes,
# it is equivalent to if we just weigh each data by 1.
print((df_train['Class'] == 0).sum()*weights[0] + (df_train['Class'] == 1).sum()*weights[1])

print((df_train['Class'] == 0).sum() + (df_train['Class'] == 1).sum())
```

```
3339.0
3339
```

SKLearn Code

```
# All right! So now you've got the idea of how to weigh classes differently.
# What does this mean for a machine learning algorithm like logistic regression?
# Different weights make it cost more to misclassify a minority than majority
class
# We can use code below to apply LR to the differently weighted datasets,
# with the extra argument class_weight='balanced'.

clf_weighted = LogisticRegression(class_weight='balanced', random_state=888)
clf_weighted.fit(df_train[features], df_train['Class'])

y_pred = clf_weighted.predict_proba(df_test[features])[:, 1]

print("The AUC score after using Weighted Logistic Regression (balanced) ...")
roc_auc_score(df_test['Class'], y_pred)
```

The AUC score for this model after using Weighted Logistic Regression (balanced) ...
0.8275030156815439

SKLearn Code

```
# Besides changing the weights of the two classes to balance them,
# We can also specify custom weights of positive and negative classes
# For example, the below code weighs class 1 by 100 times more than class 0.

clf_weighted = LogisticRegression(class_weight={0: 1, 1: 100},
random_state=888)

clf_weighted.fit(df_train[features], df_train['Class'])
y_pred = clf_weighted.predict_proba(df_test[features])[:, 1]

print("The AUC score after using Weighted Logistic Regresion (weighted) ...")
roc_auc_score(df_test['Class'], y_pred)
```

The AUC score for this model after using Weighted Logistic Regresion (weighted) ...
0.8375552874949739

Summary

Summary

- Imbalanced data occurs when the classes of the dataset are distributed unequally. It is common for machine learning classification prediction problems.
- The challenge of working with imbalanced datasets is that most machine learning techniques will ignore, and in turn have poor performance on, the minority class, although typically it is performance on the minority class that is most important.
- One approach to addressing imbalanced datasets is to oversample the minority class. The simplest approach involves duplicating examples in the minority class (Random Oversampling), although these examples don't add any new information to the model.
- Instead, new examples can be synthesized from the existing examples. This is a type of data augmentation for the minority class and is referred to as the Synthetic Minority Oversampling Technique, or SMOTE for short.