# CHAPTER 13: Imbalanced Datasets

## Introduction

In the previous chapter, *Chapter 12*, *Feature Engineering*, where we dealt with data points related to dates, we were addressing scenarios pertaining to features. In this chapter, we will deal with scenarios where the proportions of examples in the overall dataset pose challenges.

Let's revisit the dataset we dealt with in *Chapter 3*, *Binary Classification*, in which the examples pertaining to 'No' for term deposits far outnumbered the ones with 'Yes' with a ratio of 88% to 12%. We also determined that one reason for suboptimal results with a logistic regression model on that dataset was the skewed proportion of examples. Datasets like the one we analyzed in *Chapter 3*, *Binary Classification,* which are called imbalanced datasets, are very common in real-world use cases.

Some of the use cases where we encounter imbalanced datasets include the following:

Text

Description automatically generated

In all of these use cases, we can see that what we really want to detect will be minority cases. For instance, in domains such as the medical diagnosis of rare diseases, examples where rare diseases exist could even be less than 1% of the total examples. One inherent characteristic of use cases with imbalanced datasets is that the quality of the classifier is not apparent if the right metric is not used. This makes the problem of imbalanced datasets really challenging.

In this chapter, we will discuss strategies for identifying imbalanced datasets and ways to mitigate the effects of imbalanced datasets.

## Understanding The Business Context

The business head of the Insurance company for which you are working as a data scientist recently raised the alarm about the Fraud Insurance Claims. Insurance fraud is a huge problem in the industry. It's difficult to identify fraud claims. IHS is in a unique position to help the Auto Insurance industry with this problem.

In this chapter, we will be working with some auto insurance data to demonstrate how we can create a predictive model that predicts if an insurance claim is fraudulent or not. This will be a Binary Classification task, and we will be creating a Logistic Regression model.

First, we begin with an analysis of the issue.

## Exercise 13.01: Benchmarking the Logistic Regression Model on the Insurance Dataset

The dataset you will be using in this exercise can be found on our GitHub repository

https://github.com/fenago/DSBook/blob/main/Chapter%2013%20-%20Imbalanced%20datasets/datasets/insurance\_claims.csv

1. Import require packages



1. Import the dataset

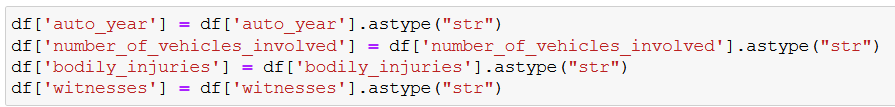
Graphical user interface

Description automatically generated with medium confidence

Table

Description automatically generated

1. Data type conversion



We have converted the variable types. Let’s the types of the variables using .info function

Graphical user interface, table

Description automatically generated

1. Normalize the numerical features (age, balance, and duration) through scaling, which was covered in *Chapter 3*, *Binary Classification*. Enter the following code:

Let’s first create a list which contains all the numeric data type variables to perform scaling on the numeric data.

Graphical user interface, application, Word

Description automatically generated

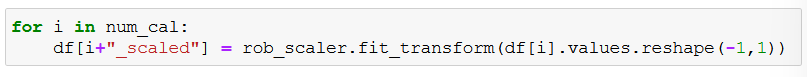
Importing the required package to scale the variables

Text

Description automatically generated

1. After scaling the numerical data, we convert each of the columns to a scaled version, as in the following code snippet:

Writing the for loop to iterate through all the numeric variables and perform future scaling.



You should get the following output:

Table

Description automatically generated

1. Now, drop the original features after we introduce the scaled features using the .drop() function:



1. Display the first five columns using the .head() function:

Graphical user interface

Description automatically generated with medium confidence

You should get the following output:

Table

Description automatically generated

1. Convert all the categorical variables to dummy variables using the .get\_dummies() function:

Let’s get all the categorical variables into list first.

Text

Description automatically generated

Text, letter

Description automatically generated

Now we can go ahead and convert the categorical variables to the dummy variables.



If you view the top 5 observation using .head() function, you should get the following output:

Table

Description automatically generated

1. Separate the numerical data and observe the shape:

A picture containing text

Description automatically generated

1. Create the independent variables, X, and dependent variables, Y, from the combined dataset for modeling, as in the following code snippet:

Graphical user interface, text, application

Description automatically generated

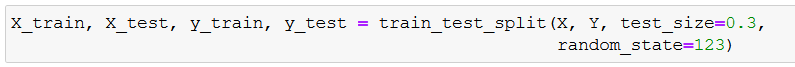
A picture containing graphical user interface

Description automatically generated

1. Now, import the necessary functions of train\_test\_split() and LogisticRegression from sklearn:

A picture containing text

Description automatically generated



1. Now, fit the model using .fit on the training data:

Text

Description automatically generated

1. Next, find the prediction on the test set and print the accuracy scores:

Text, letter

Description automatically generated

1. Now, use both the confusion\_matrix() and classification\_report() functions to generate the metrics for further analysis, which we will cover in the *Analysis of the Result* section:

Text, letter

Description automatically generated

You should get the following output:

Table

Description automatically generated

You will get metrics similar to the following. However, the values will vary due to the variability in the modeling process.

In this exercise, From the metrics, we can see that the number of values for No is relatively higher than that for Yes.

To understand more about the reasons behind the skewed results, we will analyze these metrics in detail in the following section.

## Analysis of the Result

To analyze the results obtained in the previous section, let's expand the confusion matrix in the form:

Table

Description automatically generated

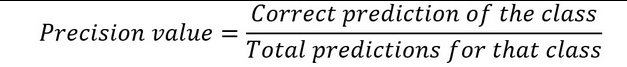
We enter the values 205, 29, 40, and 26 from the output we got from the previous exercise. We then place these values as shown in the diagram. We will represent the propensity to apply fraud claim (No) as the positive class and the other as the negative class. So, from the confusion matrix, we can calculate the accuracy measures, which were covered in Chapter 3, Binary Classification. The accuracy of the model is given by:



In our case, it will (205+26)/(205+29+40+26), or 77%

From the accuracy perspective, the model would seem like it is doing a reasonable job. However, the reality might be quite different. To find out what's really the case, let's look at the precision and recall values, which are available from the classification report we obtained. The formulae for precision for any class was covered in Chapter 3, Binary Classification

The precision value of any class is given by:



In our case, for the positive class, the precision is TP/(TP + FP), which is 205/ (205 + 29), which comes to approximately 84%.

In the case of the negative class, the precision could be written as *TN / (TN + FN)*, which is 26 / (26 + 29), which comes to approximately 47%.

Similarly, the recall value for any class can be represented as follows:



Recall indicates the ability of the classifier to correctly identify the respective classes. From the metrics, we see that the model that we built does a good job of identifying the positive classes but does a very poor job of correctly identifying the negative class.

Why do you think that the classifier is biased toward one class? The answer to this can be unearthed by looking at the class balance in the training set.

The following code will generate the percentages of the classes in the training data:

Text

Description automatically generated

From this, we can see that the majority of the training set (74%) is made up of the positive class. This imbalance is one of the major reasons behind the poor metrics that we have had with the logistic regression classifier we have selected.

Now, let's look at the challenges of imbalanced datasets.

## Challenges of Imbalanced Datasets

As seen from the classifier example, one of the biggest challenges with imbalanced datasets is the bias toward the majority class, which ended up being 74% in the previous example. This will result in suboptimal results. However, what makes such cases even more challenging is the deceptive nature of results if the right metric is not used.

Let's take, for example, a dataset where the negative class is around 99% and the positive class is 1% (as in a use case where a rare disease has to be detected, for instance).

Have a look at the following code snippet:

Graphical user interface, text, application, chat or text message, website

Description automatically generated

Suppose we had a poor classifier that was capable of only predicting the negative class; we would get the following confusion matrix:

Table

Description automatically generated

From the confusion matrix, let's calculate the accuracy measures. Have a look at the following code snippet:

A screenshot of a computer

Description automatically generated with medium confidence

With such a classifier, if we were to use a metric such as accuracy, we still would get a result of around 99%, which, in normal circumstances, would look outstanding. However, in this case, the classifier is doing a bad job. Think of the real-life impact of using such a classifier and a metric such as accuracy. The impact on patients who have rare diseases and who get wrongly classified as not having the disease could be fatal.

Therefore, it is important to identify cases with imbalanced datasets and equally important to pick the right metric for analyzing such datasets. The right metric in this example would have been to look at the recall values for both the classes:

A screenshot of a computer

Description automatically generated with medium confidence

From the recall values, we could have identified the bias of the classifier toward the majority class, prompting us to look at strategies for mitigating such biases, which is the next topic we will focus on.

## Strategies for Dealing with Imbalanced Datasets

Now that we have identified the challenges of imbalanced datasets, let's look at strategies for combatting imbalanced datasets:

Diagram

Description automatically generated

## Collecting More Data

Having encountered an imbalanced dataset, one of the first questions you need to ask is whether it is possible to get more data. This might appear naïve, but collecting more data, especially from the minority class, and then balancing the dataset should be the first strategy for addressing the class imbalance.

## Resampling Data

In many circumstances, collecting more data, especially from minority classes, can be challenging as data points for the minority class will be very minimal. In such circumstances, we need to adopt different strategies to work with our constraints and still strive to balance our dataset. One effective strategy is to resample our dataset to make the dataset more balanced. Resampling would mean taking samples from the available dataset to create a new dataset, thereby making the new dataset balanced.

Let's look at the idea in detail:

A picture containing text, air conditioner, screenshot

Description automatically generated

As seen in Figure above, the idea behind resampling is to randomly pick samples from the majority class to make the final dataset more balanced. In the diagram, we can see that the minority class has the same number of examples as the original dataset and that the majority class is under-sampled to make the final dataset more balanced. Resampling examples of this type is called random undersampling as we are undersampling the majority class. We will perform random undersampling in the following exercise.

## Exercise 13.02: Implementing Random Undersampling and Classification on Our Insurance Dataset to Find the Optimal Result

In this exercise, you will undersample the majority class (propensity 'No') and then make the dataset balanced. On the new balanced dataset, you will fit a logistic regression model and then analyze the results:

1. Perform the initial 12 steps of Exercise 13.01, Benchmarking the Logistic Regression Model on the Dataset, such that the dataset is split into training and testing sets.
2. Now, join the X and y variables for the training set before resampling:

Text

Description automatically generated with medium confidence

1. Now, display the new data with the .head() function:



Graphical user interface, table

Description automatically generated

The preceding output shows some of the columns of the dataset.

Now, let's move onto separating the minority and majority classes into separate datasets.

What we will do next is separate the minority class and the majority class. This is required because we have to sample separately from the majority class to make a balanced dataset. To separate the minority class, we have to identify the indexes of the dataset where the dataset has 'yes.' The indexes are identified using .index() function.

Once those indexes are identified, they are separated from the main dataset using the .loc() function and stored in a new variable for the minority class. The shape of the minority dataset is also printed. A similar process is followed for the majority class and, after these two steps, we have two datasets: one for the minority class and one for the majority class.

1. Next, find the indexes of the sample dataset where the propensity is yes:

Chart

Description automatically generated

You should get the following output:



1. Separate by the minority class as in the following code snippet:

Text

Description automatically generated with medium confidence

You should get the following output:



1. Now, find the indexes of the majority class:

Chart

Description automatically generated

You should get the following output:



1. Separate by the majority class as in the following code snippet:

Text

Description automatically generated

Table

Description automatically generated

Once the majority class is separated, we can proceed with sampling from the majority class. Once the sampling is done, the shape of the majority class dataset and its head are printed.

Take a random sample equal to the length of the minority class to make the dataset balanced.

1. Extract the samples using the .sample() function:



The number of examples that are sampled is equal to the number of examples in the minority class. This is implemented with the parameters (n=len(ind)).

1. Now that sampling is done, the shape of the majority class dataset and its head is printed:

Graphical user interface, text

Description automatically generated

You should get the following output:

A screenshot of a computer

Description automatically generated with medium confidence

1. After preparing the individual dataset, we can now concatenate them together using the pd.concat() function:

Graphical user interface, text

Description automatically generated with medium confidence

Note

In this case, we are concatenating in the vertical direction and, therefore, axis=0 is used.

1. Now, shuffle the dataset so that both the minority and majority classes are evenly distributed using the shuffle function:

Text

Description automatically generated

You should get the following output:

Table

Description automatically generated

1. Now, separate the shuffled dataset into the independent variables, X\_trainNew, and dependent variables, y\_trainNew. The separation is to be done using the index features 0 to 51 for the dependent variables using the .iloc() function in pandas. The dependent variables are separated by sub-setting with the column name 'y':

Text, letter

Description automatically generated

You should get the following output:

Table

Description automatically generated

Now, fit the model on the new data and generate the confusion matrix and classification report for our analysis.

1. First, define the LogisticRegression function with the following code snippet:

Graphical user interface, text, application

Description automatically generated

1. Next, perform the prediction on the test with the following code snippet:

Graphical user interface, text, application

Description automatically generated

You should get the following output:



1. Now, generate the confusion matrix for the model and print the results:

Text

Description automatically generated

You should get the following output:

Table

Description automatically generated

## Analysis

Let's analyze the results and compare them with those of the benchmark logistic regression model that we built at the beginning of this chapter. In the benchmark model, we had the problem of the model being biased toward the majority class with a very low recall value for the yes cases.

Now, by balancing the dataset, we have seen that the recall for the minority class has improved, from a low of 0.39 to around 0.56. This means that by balancing the dataset, the classifier has improved its ability to identify negative cases.

However, we can see that our overall accuracy has taken a hit. From a high of around 77%, it has come down to around 66%. One major area where accuracy has taken a hit is the number of false positives, which are those No cases that were wrongly predicted as Yes.

Analyzing the result from a business perspective, this is a much better scenario than the one we got in the benchmark model. In the benchmark model, out of the total 66 Yes cases, 40 were correctly identified. However, after balancing, we were able to identify 29 out of 66 claims from the dataset were likely to fraud, which can potentially result in a weak prediction rate.

## Generating Synthetic Samples

In the previous section, we looked at the undersampling method, where we downsized the majority class to make the dataset balanced. However, when undersampling, we reduced the size of the dataset. In many circumstances, downsizing the dataset can have adverse effects on the predictive power of the classifier. An effective way to counter the downsizing of the dataset is to oversample the minority class. Oversampling is done by generating new synthetic data points similar to those of the minority class, thereby balancing the dataset.

Two very popular methods for generating such synthetic points are:

Text

Description automatically generated

The way the SMOTE algorithm generates synthetic data is by looking at the neighborhood of minority classes and generating new data points within the neighborhood:

Chart, scatter chart

Description automatically generated

Let's explain the concept of generating synthetic datasets with a pictorial representation. Let's assume that figure above represents a dataset with two classes: the grey circles represent the minority class, and the black circles represent the majority class.

In creating synthetic points, an imaginary line connecting all the minority samples in the neighborhood is created and new data points are generated on this line, as shown in *Figure below*, thereby balancing the dataset:

Chart, scatter chart

Description automatically generated

However, MSMOTE is an advancement over the SMOTE algorithm and has a different approach to generating synthetic points. MSMOTE classifies the minority class into three distinct groups: **security samples**, **border samples**, and **latent noise samples**. Different strategies are adopted to generate neighborhood points based on the group each minority class falls into.

We will see the implementation of both SMOTE and MSMOTE in the following section.

## Implementation of SMOTE and MSMOTE

SMOTE and MSMOTE can be implemented from a package called smote-variants in Python. The library can be installed through pip install in the Colab notebook as shown here:

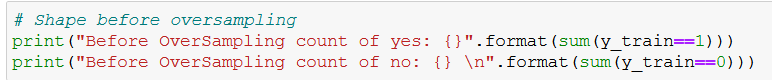


Let's now implement both these methods and analyze the results.

## Exercise 13.03: Implementing SMOTE on our Insurance Dataset to Find the Optimal Result

In this exercise, we will generate synthetic samples of the minority class using SMOTE and then make the dataset balanced. Then, on the new balanced dataset, we will fit a logistic regression model and analyze the results:

1. Implement all the steps of Exercise 13.01, Benchmarking the Logistic Regression Model on the Dataset, until the splitting of the train and test sets (Step 12).
2. Now, print the count of both the classes before we oversample:



You should get the following output:

Text, letter

Description automatically generated

Note

The counts mentioned in this output can vary because of a variability in the sampling process.

Next, we will be oversampling the training set using SMOTE.

1. Begin by importing sv and numpy:

Text

Description automatically generated

The library files that are required for oversampling the training set include the smote\_variants library, which we installed earlier. This is imported as sv. The other library that is required is numpy, as the training set will have to be given a numpy array for the smote\_variants library.

1. Now, instantiate the SMOTE library to a variable called oversampler using the sv.SMOTE() function:

Graphical user interface, text, application

Description automatically generated

This is a common way of instantiating any of the variants of SMOTE from the smote\_variants library.

1. Now, sample the process using the .sample() function of oversampler:

A picture containing text

Description automatically generated

Note

Both the X and Y variables are converted to numpy arrays before applying the .sample() function.

1. Now, print the shapes of the new X and y variables and the counts of the classes. You will note that the size of the overall dataset has increased from the earlier count of around 519 (519 + 519) to 1038. The increase in size can be attributed to the fact that the minority class has been oversampled from 181 to 519:

A picture containing text

Description automatically generated

You should get the following output:

Text

Description automatically generated

Note

The counts mentioned in this output can vary because of variability in the sampling process.

Now that we have generated synthetic points using SMOTE and balanced the dataset, let's fit a logistic regression model on the new sample and analyze the results using a confusion matrix and a classification report.

1. Define the LogisticRegression function:

Text

Description automatically generated

1. Now, predict using .predict on the test set, as mentioned in the following code snippet:



1. Next, print the accuracy values:

A picture containing text

Description automatically generated

You should get the following output:

Text

Description automatically generated with medium confidence

1. Then, generate ConfusionMatrix for the model:

Text

Description automatically generated

You should get the following output:

A picture containing text

Description automatically generated

1. Generate Classification\_report for the model:

Text

Description automatically generated

You should get the following output:

Table

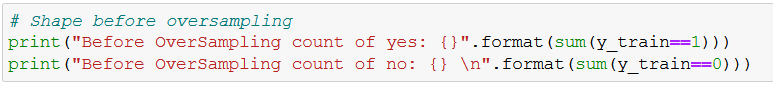
Description automatically generated with low confidence

In the next exercise, we will be implementing MSMOTE.

## Exercise 13.04: Implementing MSMOTE on our Insurance Dataset to Find the Optimal Result

In this exercise, we will generate synthetic samples of the minority class using MSMOTE and then make the dataset balanced. Then, on the new balanced dataset, we will fit a logistic regression model and analyze the results. This exercise will be very similar to the previous one.

1. Implement all the steps of Exercise 13.01, Benchmarking the Logistic Regression Model on the Dataset, until the splitting of the train and test sets (Step *12*).
2. Now, print the count of both the classes before we oversample:



You should the following output:

Text

Description automatically generated

1. Now, instantiate the MSMOTE library to a variable called oversampler using the sv.MSMOTE() function:

Graphical user interface, text, application

Description automatically generated

1. Now, sample the process using the .sample() function of oversampler:

A picture containing text

Description automatically generated

Note

Both the X and y variables are converted to numpy arrays before applying the .sample() function.

Now, print the shapes of the new X and y variables and also the counts of the classes:

A picture containing text

Description automatically generated

You should get the following output:

Text, letter

Description automatically generated

Now that we have generated synthetic points using MSMOTE and balanced the dataset, let's fit a logistic regression model on the new sample and analyze the results using a confusion matrix and a classification report.

1. Define the LogisticRegression function:

Text

Description automatically generated

1. Now, predict using .predict on the test set as in the following code snippet:



You should get the following output:



1. Generate the ConfusionMatrix for the model:

Text

Description automatically generated

You should get the following output:

A picture containing text

Description automatically generated

1. Generate the Classification\_report for the model:

Text

Description automatically generated

You should get the following output:

Table

Description automatically generated

From the implementation of MSMOTE, it is seen that the metrics have improved compared to the SMOTE implementation from Exercise 13.03, Implementing SMOTE on Our Insurance Dataset to Find the Optimal Result. However, the result with base model seems to be similar. We can then conclude that MSMOTE might not be the best method for this use case.

## Applying Balancing Techniques on a HealthCare Dataset

Now that we have seen different balancing techniques, let's apply these techniques to a new dataset that is related to the healthcare. This dataset is available at the following link:

https://github.com/fenago/DSBook/blob/main/Chapter%2013%20-%20Imbalanced%20datasets/datasets/healthcare-dataset-stroke-data.csv

This dataset has various variables related to the heart stroke, Hypertension, heart diseases, as so on.

The problem statement is to predict whether a person will get a heart stroke. This dataset is a imbalanced one, with the cases where people are not health being the minority. You will be using this dataset in the following activity.

## Activity 13.01: Finding the Best Balancing Technique by Fitting a Classifier on the HealthCare Dataset

You are working as a data scientist for a healthcare company. You have encountered a dataset that is highly imbalanced, and you want to correct the class imbalance before fitting the classifier to analyze the heart stroke. You know different methods for correcting the imbalance in datasets and you want to compare them to find the best method before fitting the model.

In this activity, you need to implement all of the three methods that you have come across so far and compare the results.

Use the MinMaxscaler function to scale the dataset instead of the robust scaler function you have been using so far. Compare the methods based on the results you get by fitting a logistic regression model on the dataset.

The steps are as follows:

1. Implement all the initial steps, which include installing smote-variants and loading the data using pandas.
2. Normalize the numerical raw data using the MinMaxScaler() function we learned about in Chapter 3, Binary Classification.
3. Create dummy data for the categorical variables using the pd.get\_dummies() function.
4. Separate the numerical data from the original data frame.
5. Concatenate numerical data and dummy categorical data using the pd.concat() function.
6. Split the earlier dataset into train and test sets using the train\_test\_split() function.

Since the dataset is imbalanced, you need to perform the various techniques mentioned in the following steps.

1. For the undersampling method, find the index of the minority class using the .index() function and separate the minority class. After that, sample the majority class and make the majority dataset equal to the minority class using the .sample() function. Concatenate both the minority and under-sampled majority class to form a new dataset. Shuffle the dataset and separate the X and Y variables.
2. Fit a logistic regression model on the under-sampled dataset and name it model1.
3. For the SMOTE method, create the oversampler using the sv.SMOTE() function and create the new X and Y training sets.
4. Fit a logistic regression model using SMOTE and name it model2.
5. Import the smote-variant library and instantiate the MSMOTE algorithm using the sv.MSMOTE() function.
6. Create the oversampled data using the oversampler. Please note that the X and y variables have to be converted to a numpy array before oversampling
7. Fit the logistic regression model using the MSMOTE dataset and name the model model3.
8. Generate the three separate predictions for each model.
9. Generate separate accuracy metrics, classification reports, and confusion matrices for each of the predictions.
10. Analyze the results and select the best method.

## Summary

In this chapter, we learned about imbalanced datasets and strategies for addressing imbalanced datasets. We introduced the use cases where imbalanced datasets would be encountered. We looked at the challenges posed by imbalanced datasets and we were introduced to the metrics that should be used in the case of an imbalanced dataset. We formulated strategies for dealing with imbalanced datasets and implemented different strategies, such as random undersampling and oversampling, for balancing datasets. We then fit different models after balancing the datasets and analyzed the results.

Balancing datasets is a very effective way to improve the performance of your classifiers. However, it should be noted that there could be a degradation of overall accuracy measures for the majority class due to balancing. What strategies to adopt in what situations should be arrived at based on the problem statement and also after rigorous experiments for those problem statements.

Having learned about methods for dealing with imbalanced datasets, we will now be introduced to another important technique that is prevalent in many modern datasets called dimensionality reduction. Different techniques for dimensionality reduction will be addressed in *Chapter 14*, *Dimensionality Reduction*.