Data Quality Considerations

The quality of data used in any machine learning problem, supervised or unsupervised, is critical to the performance of the final model, and should be at the forefront when planning any machine learning project. As a simple rule of thumb, if you have clean data, in sufficient quantity, with a good correlation between the input data type and the desired output, then the specifics regarding the type and details of the selected supervised learning model become significantly less important in achieving a good result.

In reality, however, this is rarely the case. There are usually some issues regarding the quantity of available data, the quality or **signal-to-noise ratio** in the data, the correlation between the input and output, or some combination of all three factors. As such, we will use this last section of this chapter to consider some of the data quality problems that may occur and some mechanisms for addressing them. Previously, we mentioned that in any machine learning problem, having a thorough understanding of the dataset is critical if we to are construct a high-performing model. This is particularly the case when looking into data quality and attempting to address some of the issues present within the data. Without a comprehensive understanding of the dataset, additional noise or other unintended issues may be introduced during the data cleaning process, leading to further degradation of performance.

> **Note**: A detailed description of the Titanic dataset and the type of data included is contained in the *Loading Data in pandas* section. If you need a quick refresher, go back and review that section now.

Managing Missing Data

As we discussed earlier, the ability of pandas to read data with missing values is both a blessing and a curse and, arguably, is the most common issue that needs to be managed before we can continue with developing our supervised learning model. The simplest, but not necessarily the most effective, method is to just remove or ignore those entries that are missing data. We can easily do this in pandas using the `dropna` method on the DataFrame:

```
complete_data = df.dropna()
```

There is one very significant consequence of simply dropping rows with missing data and that is we may be throwing away a lot of important information. This is highlighted very clearly in the Titanic dataset as a lot of rows contain missing data. If we were to simply ignore these rows, we would start with a sample size of 1,309 and end with a sample of 183 entries. Developing a reasonable supervised learning model with a little over 10% of the data would be very difficult indeed. The following code displays the use of the `dropna()` method to handle the missing entries:

```
len(df)
```

The preceding input produces the following output:

```
1309
```

The `dropna()` method is implemented as follows:

```
len(df.dropna())
```

The preceding input produces the following output:

```
183
```

So, with the exception of the early, explorative phase, it is rarely acceptable to simply discard all rows with invalid information. We can identify which rows are actually missing information and whether the missing information is a problem unique to certain columns or is consistent throughout all columns of the dataset. We can use **aggregate** to help us here as well:

```
df.aggregate(lambda x: x.isna().sum() )
```

The output will be as follows:

```
Cabin       1014
Embarked       2
Fare           1
Pclass         0
Ticket         0
Age          263
Name           0
Parch          0
Sex            0
SibSp          0
Survived     418
dtype: int64
```

Now, this is useful! We can see that the vast majority of missing information is in the **Cabin** column, some in **Age**, and a little more in **Survived**. This is one of the first times in the data cleaning process that we may need to make an educated judgment call.

What do we want to do with the **Cabin** column? There is so much missing information here that, in fact, it may not be possible to use it in any reasonable way. We could attempt to recover the information by looking at the names, ages, and number of parents/siblings and see whether we can match some families together to provide information, but there would be a lot of uncertainty in this process. We could also simplify the column by using the level of the cabin on the ship rather than the exact cabin number, which may then correlate better with name, age, and social status. This is unfortunate as there could be a good correlation between **Cabin** and **Survived**, as perhaps those passengers in the lower decks of the ship may have had a harder time evacuating. We could examine only the rows with valid **Cabin** values to see whether there is any predictive power in the **Cabin** entry; but, for now, we will simply disregard **Cabin** as a reasonable input (or feature).

We can see that the **Embarked** and **Fare** columns only have three missing samples between them. If we decided that we needed the **Embarked** and **Fare** columns for our model, it would be a reasonable argument to simply drop these rows. We can do this using our indexing techniques, where **~** represents the **not** operation, or flipping the result (that is, where **df.Embarked** is not **NaN** and **df.Fare** is not **NaN**):

```
df_valid = df.loc[(~df.Embarked.isna()) & (~df.Fare.isna())]
```

The missing age values are a little more interesting, as there are too many rows with missing age values to just discard them. But we also have a few more options here, as we can have a little more confidence in some plausible values to fill in. The simplest option would be to simply fill in the missing age values with the mean age for the dataset:

```
df_valid[['Age']] = df_valid[['Age']].fillna(df_valid.Age.mean())
```

This is okay, but there are probably better ways of filling in the data rather than just giving all 263 people the same value. Remember, we are trying to clean up the data with the goal of maximizing the predictive power of the input features and the survival rate. Giving everyone the same value, while simple, doesn't seem too reasonable. What if we were to look at the average ages of the members of each of the classes (**Pclass**)? This may give a better estimate, as the average age reduces from class 1 through 3, as you can see in the following code:

```
df_valid.loc[df.Pclass == 1, 'Age'].mean()
```

The preceding input produces the following output:

```
37.956806510096975
```

Average age for class 2 is as follows:

```
df_valid.loc[df.Pclass == 2, 'Age'].mean()
```

The preceding input produces the following output:

```
29.52440879717283
```

Average age for class 3 is as follows:

```
df_valid.loc[df.Pclass == 3, 'Age'].mean()
```

The preceding input produces the following output:

```
26.23396338788047
```

What if we were to consider the sex of the person as well as ticket class (social status)? Do the average ages differ here too? Let's find out:

```
for name, grp in df_valid.groupby(['Pclass', 'Sex']):
    print('%i' % name[0], name[1], '%0.2f' % grp['Age'].mean())
```

The output will be as follows:

```
1 female 36.84
1 male 41.03
2 female 27.50
2 male 30.82
3 female 22.19
3 male 25.86
```

We can see here that males in all ticket classes are typically older. This combination of sex and ticket class provides much more resolution than simply filling in all missing fields with the mean age. To do this, we will use the **`transform`** method, which applies a function to the contents of a series or DataFrame and returns another series or DataFrame with the transformed values. This is particularly powerful when combined with the **`groupby`** method:

```
mean_ages = df_valid.groupby(['Pclass', 'Sex'])['Age'].\
    transform(lambda x: x.fillna(x.mean()))
df_valid.loc[:, 'Age'] = mean_ages
```

There is a lot in these two lines of code, so let's break them down into components. Let's look at the first line:

```
mean_ages = df_valid.groupby(['Pclass', 'Sex'])['Age'].\
    transform(lambda x: x.fillna(x.mean()))
```

We are already familiar with **`df_valid.groupby(['Pclass', 'Sex'])['Age']`**, which groups the data by ticket class and sex and returns only the **`Age`** column. The **`lambda x: x.fillna(x.mean())`** Lambda function takes the input pandas series and fills the **`NaN`** values with the mean value of the series.

The second line assigns the filled values within **`mean_ages`** to the **`Age`** column. Note the use of the **`loc[:, 'Age']`** indexing method, which indicates that all rows within the **`Age`** column are to be assigned the values contained within **`mean_ages`**:

```
df_valid.loc[:, 'Age'] = mean_ages
```

We have described a few different ways of filling in the missing values within the `Age`column, but by no means has this been an exhaustive discussion. There are many more methods that we could use to fill the missing data: we could apply random values within one standard deviation of the mean for the grouped data, and we could also look at grouping the data by sex and the number of parents/children (`Parch`) or by the number of siblings, or by ticket class, sex, and the number of parents/children. What is most important about the decisions made during this process is the end result of the prediction accuracy. We may need to try different options, rerun our models and consider the effect on the accuracy of final predictions. This is an important aspect of the process of feature engineering, that is, selecting the features or components that provide the model with the most predictive power. You will find that, during this process, you will try a few different features, run the model, look at the end result, and repeat this process until you are happy with the performance.

The ultimate goal of this supervised learning problem is to predict the survival of passengers on the Titanic given the information we have available. So, that means that the `Survived` column provides our labels for training. What are we going to do if we are missing 418 of the labels? If this was a project where we had control over the collection of the data and access to its origins, we would obviously correct this by recollecting or asking for the labels to be clarified. With the Titanic dataset, we do not have this ability so we must make another educated judgment call. One approach would be to drop those rows from the training data, and later use a model trained on the (smaller) training set to predict the outcome for the others (this is, in fact, the task given in the Kaggle Titanic competition). In some business problems, we may not have the option of simply ignoring these rows; we might be trying to predict future outcomes of a very critical process and this data is all we have. We could try some unsupervised learning techniques to see whether there are some patterns in the survival information that we could use. However, by estimating the ground truth labels by means of unsupervised techniques, we may introduce significant noise into the dataset, reducing our ability to accurately predict survival.

Class Imbalance

Missing data is not the only problem that may be present within a dataset. Class imbalance – that is, having more of one class or classes compared to another – can be a significant problem, particularly in the case of classification problems (we'll see more on classification in *Chapter 5*, *Classification Techniques*), where we are trying to predict which class (or classes) a sample is from. Looking at our `Survived` column, we can see that there are far more people who perished (`Survived` equals `0`) than survived (`Survived`equals `1`) in the dataset, as you can see in the following code:

```
len(df.loc[df.Survived ==1])
```

The output is as follows:

```
342
```

The number of people who perished are:

```
len(df.loc[df.Survived ==0])
```

The output is as follows:

```
549
```

If we don't take this class imbalance into account, the predictive power of our model could be significantly reduced as, during training, the model would simply need to guess that the person did not survive to be correct 61% (*549 / (549 + 342)*) of the time. If, in reality, the actual survival rate was, say, 50%, then when being applied to unseen data, our model would predict *did not survive* too often.

There are a few options available for managing class imbalance, one of which, similar to the missing data scenario, is to randomly remove samples from the over-represented class until balance has been achieved. Again,

this option is not ideal, or perhaps even appropriate, as it involves ignoring available data. A more constructive example may be to oversample the under-represented class by randomly copying samples from the under-represented class in the dataset to boost the number of samples. While removing data can lead to accuracy issues due to discarding useful information, oversampling the under-represented class can lead to being unable to predict the label of unseen data, also known as overfitting (which we will cover in *Chapter 6*, *Ensemble Modeling*).

Adding some random noise to the input features for oversampled data may prevent some degree of overfitting, but this is highly dependent on the dataset itself. As with missing data, it is important to check the effect of any class imbalance corrections on the overall model performance. It is relatively straightforward to copy more data into a DataFrame using the `append` method, which works in a very similar fashion to lists. If we wanted to copy the first row to the end of the DataFrame, we would do this:

```
df_oversample = df.append(df.iloc[0])
```

Low Sample Size

The field of machine learning can be considered a branch of the larger field of statistics. As such, the principles of confidence and sample size can also be applied to understand the issues with a small dataset. Recall that if we were to take measurements from a data source with high variance, then the degree of uncertainty in the measurements would also be high and more samples would be required to achieve a specified confidence in the value of the mean. The sample principles can be applied to machine learning datasets. Those datasets with a variance in the features with the most predictive power generally require more samples for reasonable performance as more confidence is also required.

There are a few techniques that can be used to compensate for a reduced sample size, such as transfer learning. However, these lie outside the scope of this course. Ultimately, though, there is only so much that can be done with a small dataset, and significant performance increases may only occur once the sample size is increased.