3.2 Common Data Preparation Tasks

We can define data preparation as the transformation of raw data into a form that is more suitable for modeling. Nevertheless, there are steps in a predictive modeling project before and after the data preparation step that are important and inform the data preparation that is to be performed. The process of applied machine learning consists of a sequence of steps (introduced in Chapter 1). We may jump back and forth between the steps for any given project, but all projects have the same general steps; they are:

- **Step 1**: Define Problem.
- **Step 2**: Prepare Data.
- **Step 3**: Evaluate Models.
- **Step 4**: Finalize Model.

We are concerned with the data preparation step (Step 2), and there are common or standard tasks that you may use or explore during the data preparation step in a machine learning project. The types of data preparation performed depend on your data, as you might expect. Nevertheless, as you work through multiple predictive modeling projects, you see and require the same types of data preparation tasks again and again.

These tasks include:

- **Data Cleaning**: Identifying and correcting mistakes or errors in the data.
- **Feature Selection**: Identifying those input variables that are most relevant to the task.
- **Data Transforms**: Changing the scale or distribution of variables.
- **Feature Engineering**: Deriving new variables from available data.
- **Dimensionality Reduction**: Creating compact projections of the data.

This provides a rough framework that we can use to think about and navigate different data preparation algorithms we may consider on a given project with structured or tabular data. Let's take a closer look at each in turn.

3.3 Data Cleaning

Data cleaning involves fixing systematic problems or errors in *messy* data. The most useful data cleaning involves deep domain expertise and could involve identifying and addressing specific observations that may be incorrect. There are many reasons data may have incorrect values, such as being mistyped, corrupted, duplicated, and so on. Domain expertise may allow obviously erroneous observations to be identified as they are different from what is expected, such as a person's height of 200 feet.

Once messy, noisy, corrupt, or erroneous observations are identified, they can be addressed. This might involve removing a row or a column. Alternately, it might involve replacing observations with new values. As such, there are general data cleaning operations that can be performed, such as:

- Using statistics to define normal data and identify outliers (Chapter 6.
- Identifying columns that have the same value or no variance and removing them (Chapter 5).
- Identifying duplicate rows of data and removing them (Chapter 5).
- Marking empty values as missing (Chapter 7).
- Imputing missing values using statistics or a learned model (Chapters 8, 9 and 10).

Data cleaning is an operation that is typically performed first, prior to other data preparation operations.

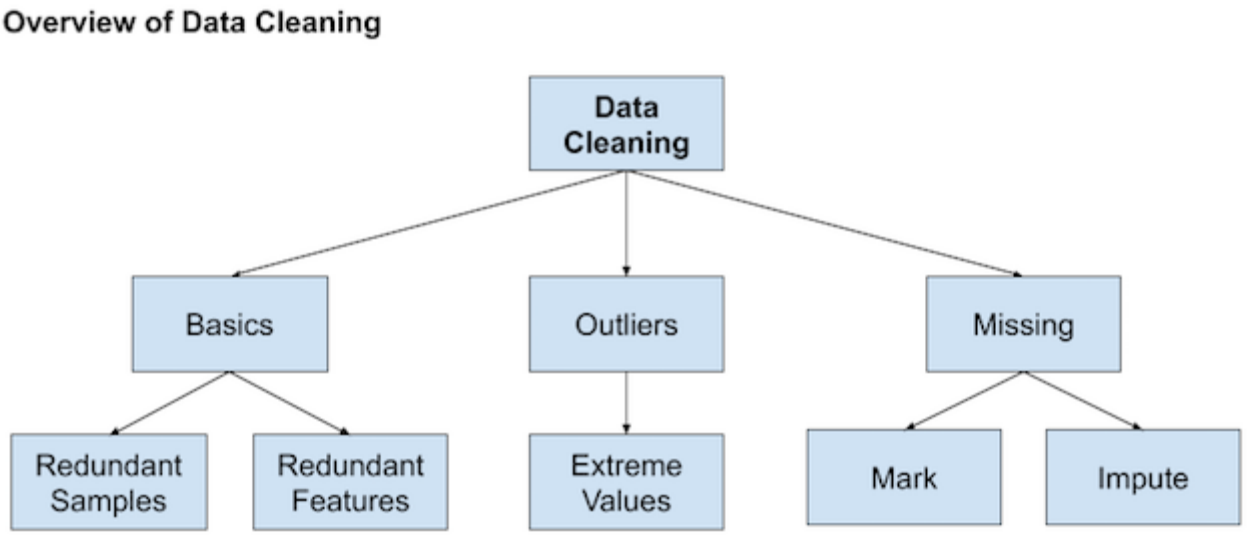

Figure 3.1: Overview of Data Cleaning Techniques.

3.4 Feature Selection

Feature selection refers to techniques for selecting a subset of input features that are most relevant to the target variable that is being predicted. This is important as irrelevant and redundant input variables can distract or mislead learning algorithms possibly resulting in lower predictive performance. Additionally, it is desirable to develop models only using the data that is required to make a prediction, e.g. to favor the simplest possible well performing model.

Feature selection techniques may generally grouped into those that use the target variable (supervised) and those that do not (unsupervised). Additionally, the supervised techniques can be further divided into models that automatically select features as part of fitting the model (intrinsic), those that explicitly choose features that result in the best performing model (wrapper) and those that score each input feature and allow a subset to be selected (filter).

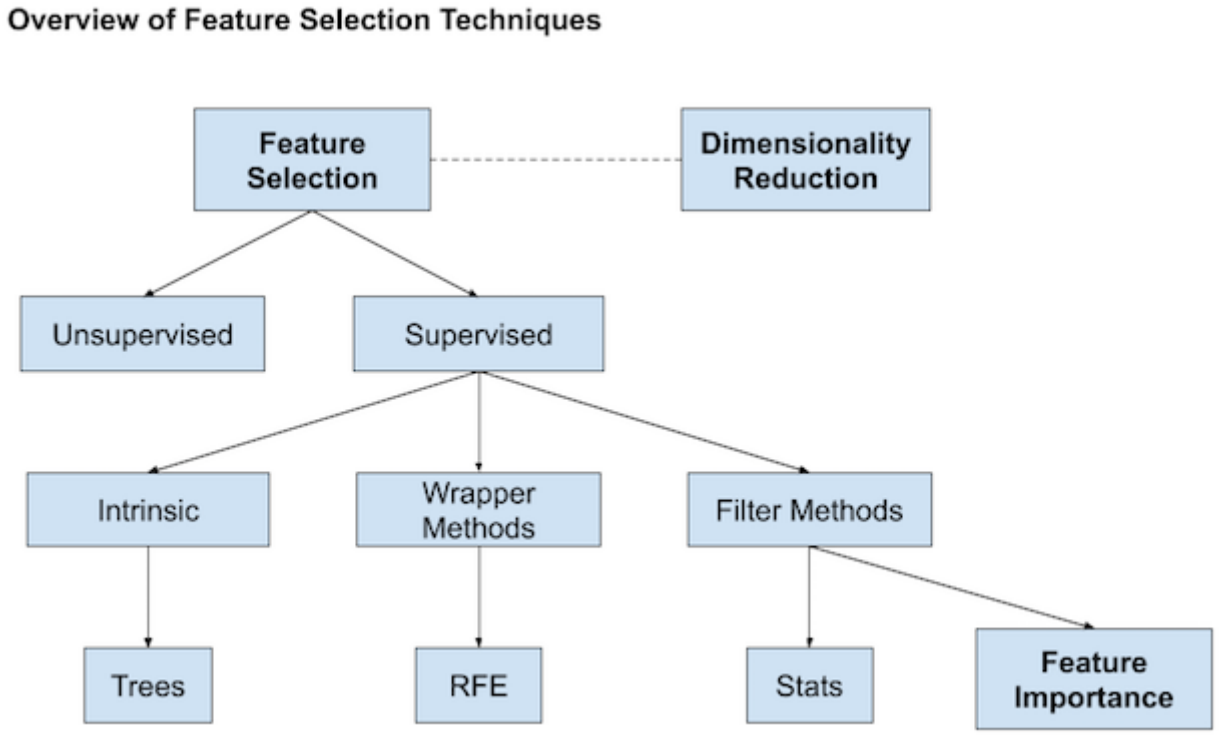

Figure 3.2: Overview of Feature Selection Techniques.

Statistical methods, such as correlation, are popular for scoring input features. The features can then be ranked by their scores and a subset with the largest scores used as input to a model. The choice of statistical measure depends on the data types of the input variables and a review of different statistical measures that can be used is introduced in Chapter 11. Additionally, there are different common feature selection use cases we may encounter in a predictive modeling project, such as:

- Categorical inputs for a classification target variable (Chapter 12).
- Numerical inputs for a classification target variable (Chapter 13).
- Numerical inputs for a regression target variable (Chapter 14).

When a mixture of input variable data types is present, different filter methods can be used. Alternately, a wrapper method such as the popular Recursive Feature Elimination (RFE) method can be used that is agnostic to the input variable type. We will explore using RFE for feature selection in Chapter 11. The broader field of scoring the relative importance of input features is referred to as feature importance and many model-based techniques exist whose outputs can be used to aide in interpreting the model, interpreting the dataset, or in selecting features for modeling. We will explore feature importance in Chapter 16.

3.5 Data Transforms

Data transforms are used to change the type or distribution of data variables. This is a large umbrella of different techniques and they may be just as easily applied to input and output variables. Recall that data may have one of a few types, such as numeric or categorical, with subtypes for each, such as integer and real-valued floating point values for numeric, and nominal, ordinal, and boolean for categorical.

- **Numeric Data Type**: Number values.
  - **Integer**: Integers with no fractional part.
  - **Float**: Floating point values.
- **Categorical Data Type**: Label values.
  - **Ordinal**: Labels with a rank ordering.
  - **Nominal**: Labels with no rank ordering.
  - **Boolean**: Values True and False.

The figure below provides an overview of this same breakdown of high-level data types.

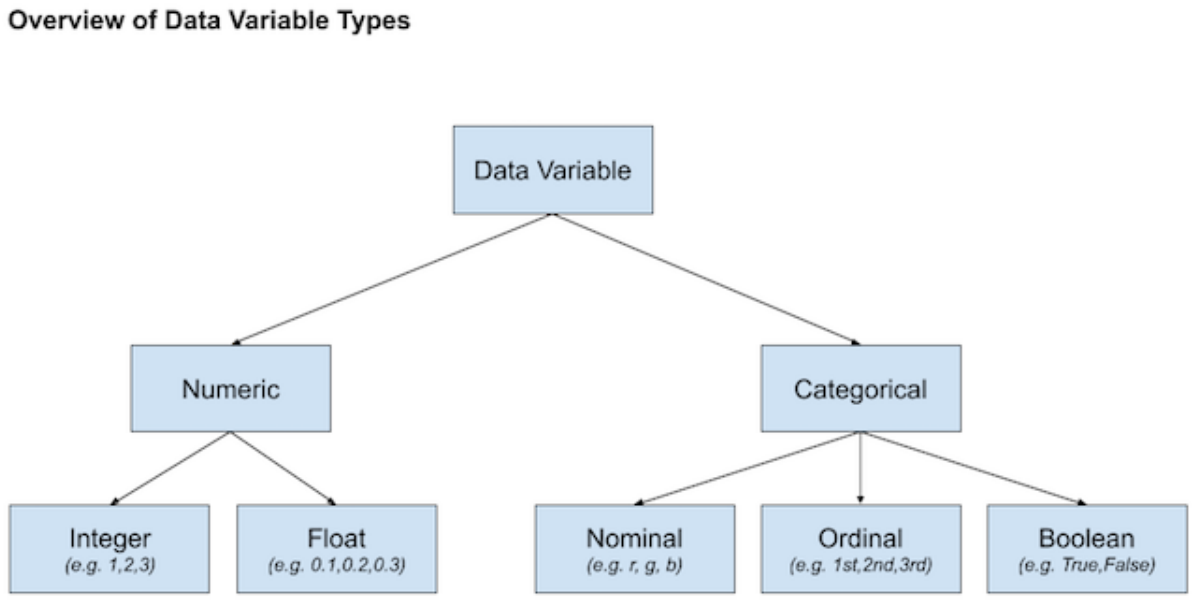

Figure 3.3: Overview of Data Variable Types.

We may wish to convert a numeric variable to an ordinal variable in a process called discretization. Alternatively, we may encode a categorical variable as integers or boolean variables, required on most classification tasks.

- **Discretization Transform**: Encode a numeric variable as an ordinal variable (Chapter 22).

- **Ordinal Transform**: Encode a categorical variable into an integer variable (Chapter 19).
- **One Hot Transform**: Encode a categorical variable into binary variables (Chapter 19).

For real-valued numeric variables, the way they are represented in a computer means there is dramatically more resolution in the range 0-1 than in the broader range of the data type. As such, it may be desirable to scale variables to this range, called normalization. If the data has a Gaussian probability distribution, it may be more useful to shift the data to a standard Gaussian with a mean of zero and a standard deviation of one.

- **Normalization Transform**: Scale a variable to the range 0 and 1 (Chapters 17 and 18).
- **Standardization Transform**: Scale a variable to a standard Gaussian (Chapter 17).

The probability distribution for numerical variables can be changed. For example, if the distribution is nearly Gaussian, but is skewed or shifted, it can be made more Gaussian using a power transform. Alternatively, quantile transforms can be used to force a probability distribution, such as a uniform or Gaussian on a variable with an unusual natural distribution.

- **Power Transform**: Change the distribution of a variable to be more Gaussian (Chapter 20).
- **Quantile Transform**: Impose a probability distribution such as uniform or Gaussian (Chapter 21).

An important consideration with data transforms is that the operations are generally performed separately for each variable. As such, we may want to perform different operations on different variable types. We may also want to use the transform on new data in the future. This can be achieved by saving the transform objects to file along with the final model trained on all available data.

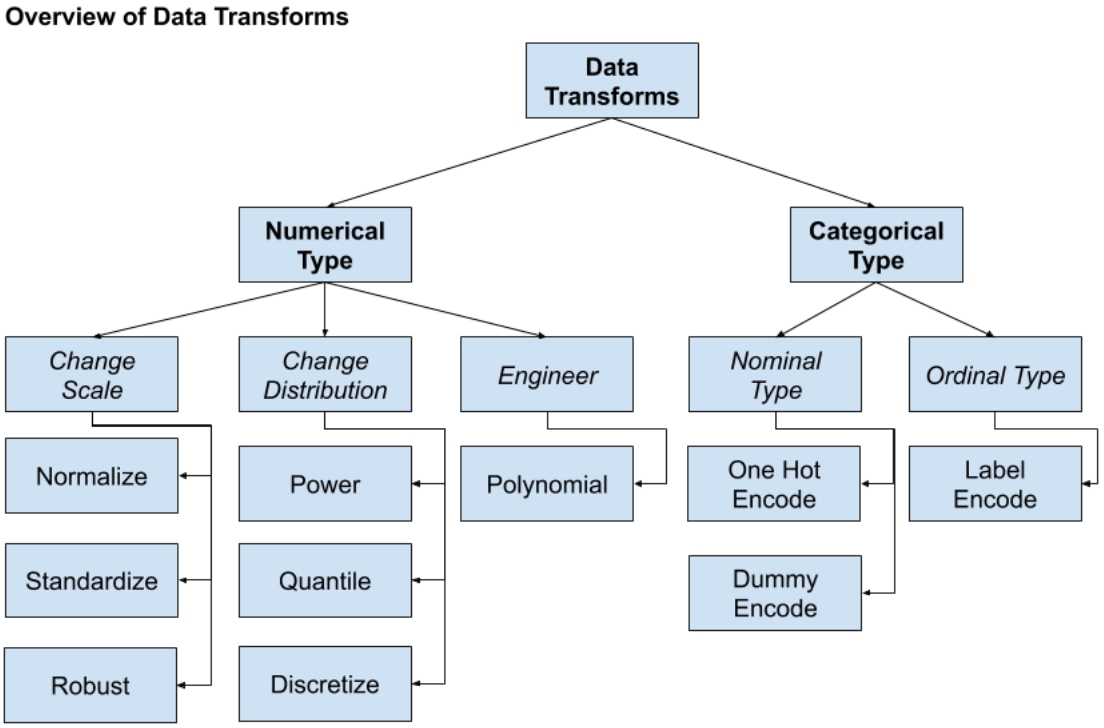

Figure 3.4: Overview of Data Transform Techniques.

3.6 Feature Engineering

Feature engineering refers to the process of creating new input variables from the available data. Engineering new features is highly specific to your data and data types. As such, it often requires the collaboration of a subject matter expert to help identify new features that could be constructed from the data. This specialization makes it a challenging topic to generalize to general methods. Nevertheless, there are some techniques that can be reused, such as:

- Adding a boolean flag variable for some state.
- Adding a group or global summary statistic, such as a mean.
- Adding new variables for each component of a compound variable, such as a date-time.

A popular approach drawn from statistics is to create copies of numerical input variables that have been changed with a simple mathematical operation, such as raising them to a power or multiplied with other input variables, referred to as polynomial features.

- **Polynomial Transform**: Create copies of numerical input variables that are raised to a power (Chapter 23).

The theme of feature engineering is to add broader context to a single observation or decompose a complex variable, both in an effort to provide a more straightforward perspective on the input data. I like to think of feature engineering as a type of data transform, although it would be just as reasonable to think of data transforms as a type of feature engineering.

Chapter 5

Basic Data Cleaning

Data cleaning is a critically important step in any machine learning project. In tabular data, there are many different statistical analysis and data visualization techniques you can use to explore your data in order to identify data cleaning operations you may want to perform. Before jumping to the sophisticated methods, there are some very basic data cleaning operations that you probably should perform on every single machine learning project. These are so basic that they are often overlooked by seasoned machine learning practitioners, yet are so critical that if skipped, models may break or report overly optimistic performance results. In this tutorial, you will discover basic data cleaning you should always perform on your dataset. After completing this tutorial, you will know:

- How to identify and remove column variables that only have a single value.
- How to identify and consider column variables with very few unique values.
- How to identify and remove rows that contain duplicate observations.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into seven parts; they are:

1. Messy Datasets
2. Identify Columns That Contain a Single Value
3. Delete Columns That Contain a Single Value
4. Consider Columns That Have Very Few Values
5. Remove Columns That Have A Low Variance
6. Identify Rows that Contain Duplicate Data
7. Delete Rows that Contain Duplicate Data

5.2 Messy Datasets

Data cleaning refers to identifying and correcting errors in the dataset that may negatively impact a predictive model.

> Data cleaning is used to refer to all kinds of tasks and activities to detect and repair errors in the data.
>
> — Page xiii, *Data Cleaning*, 2019.

Although critically important, data cleaning is not exciting, nor does it involve fancy techniques. Just a good knowledge of the dataset.

> Cleaning up your data is not the most glamourous of tasks, but it's an essential part of data wrangling. [...] Knowing how to properly clean and assemble your data will set you miles apart from others in your field.
>
> — Page 149, *Data Wrangling with Python*, 2016.

There are many types of errors that exist in a dataset, although some of the simplest errors include columns that don't contain much information and duplicated rows. Before we dive into identifying and correcting messy data, let's define some messy datasets. We will use two datasets as the basis for this tutorial, the oil spill dataset and the iris flowers dataset.

5.2.1 Oil Spill Dataset

The so-called *oil spill* dataset is a standard machine learning dataset. The task involves predicting whether the patch contains an oil spill or not, e.g. from the illegal or accidental dumping of oil in the ocean, given a vector that describes the contents of a patch of a satellite image. There are 937 cases. Each case is comprised of 48 numerical computer vision derived features, a patch number, and a class label. The normal case is no oil spill assigned the class label of 0, whereas an oil spill is indicated by a class label of 1. There are 896 cases for no oil spill and 41 cases of an oil spill. You can learn more about the dataset here:

- Oil Spill Dataset (`oil-spill.csv`).[1]
- Oil Spill Dataset Description (`oil-spill.names`).[2]

Review the contents of the data file. We can see that the first column contains integers for the patch number. We can also see that the computer vision derived features are real-valued with differing scales such as thousands in the second column and fractions in other columns. This dataset contains columns with very few unique values that provides a good basis for data cleaning.

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/oil-spill.csv
[2] https://raw.githubusercontent.com/jbrownlee/Datasets/master/oil-spill.names

5.2.2 Iris Flowers Dataset

The so-called *iris flowers* dataset is another standard machine learning dataset. The dataset involves predicting the flower species given measurements of iris flowers in centimeters. It is a multiclass classification problem. The number of observations for each class is balanced. There are 150 observations with 4 input variables and 1 output variable. You can access the entire dataset here:

- Iris Flowers Dataset (`iris.csv`).[3]
- Iris Flowers Dataset Description (`iris.names`).[4]

Review the contents of the file. The first few lines of the file should look as follows:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
...
```

Listing 5.1: Sample of the iris flowers dataset.

We can see that all four input variables are numeric and that the target class variable is a string representing the iris flower species. This dataset contains duplicate rows that provides a good basis for data cleaning.

5.3 Identify Columns That Contain a Single Value

Columns that have a single observation or value are probably useless for modeling. These columns or predictors are referred to zero-variance predictors as if we measured the variance (average value from the mean), it would be zero.

> When a predictor contains a single value, we call this a zero-variance predictor because there truly is no variation displayed by the predictor.
>
> — Page 96, *Feature Engineering and Selection*, 2019.

Here, a single value means that each row for that column has the same value. For example, the column $X1$ has the value 1.0 for all rows in the dataset:

```
X1
1.0
1.0
1.0
1.0
1.0
...
```

Listing 5.2: Example of a column that contains a single value.

[3] https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv
[4] https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.names

Columns that have a single value for all rows do not contain any information for modeling. Depending on the choice of data preparation and modeling algorithms, variables with a single value can also cause errors or unexpected results. You can detect rows that have this property using the `unique()` NumPy function that will report the number of unique values in each column. The example below loads the oil-spill classification dataset that contains 50 variables and summarizes the number of unique values for each column.

```
# summarize the number of unique values for each column using numpy
from numpy import loadtxt
from numpy import unique
# load the dataset
data = loadtxt('oil-spill.csv', delimiter=',')
# summarize the number of unique values in each column
for i in range(data.shape[1]):
  print(i, len(unique(data[:, i])))
```

Listing 5.3: Example reporting the number of unique values in each column.

Running the example loads the dataset directly and prints the number of unique values for each column. We can see that column index 22 only has a single value and should be removed.

```
0 238
1 297
2 927
3 933
4 179
5 375
6 820
7 618
8 561
9 57
10 577
11 59
12 73
13 107
14 53
15 91
16 893
17 810
18 170
19 53
20 68
21 9
22 1
23 92
24 9
25 8
26 9
27 308
28 447
29 392
30 107
31 42
32 4
33 45
34 141
```

```
35 110
36 3
37 758
38 9
39 9
40 388
41 220
42 644
43 649
44 499
45 2
46 937
47 169
48 286
49 2
```

Listing 5.4: Example output from reporting the number of unique values in each column.

A simpler approach is to use the `nunique()` Pandas function that does the hard work for you. Below is the same example using the Pandas function.

```
# summarize the number of unique values for each column using numpy
from pandas import read_csv
# load the dataset
df = read_csv('oil-spill.csv', header=None)
# summarize the number of unique values in each column
print(df.nunique())
```

Listing 5.5: Example a simpler approach to reporting the number of unique values in each column.

Running the example, we get the same result, the column index, and the number of unique values for each column.

```
0     238
1     297
2     927
3     933
4     179
5     375
6     820
7     618
8     561
9      57
10    577
11     59
12     73
13    107
14     53
15     91
16    893
17    810
18    170
19     53
20     68
21      9
22      1
```

```
23     92
24      9
25      8
26      9
27    308
28    447
29    392
30    107
31     42
32      4
33     45
34    141
35    110
36      3
37    758
38      9
39      9
40    388
41    220
42    644
43    649
44    499
45      2
46    937
47    169
48    286
49      2
dtype: int64
```

Listing 5.6: Example output from a simpler approach to reporting the number of unique values in each column.

5.4 Delete Columns That Contain a Single Value

Variables or columns that have a single value should probably be removed from your dataset

> ... simply remove the zero-variance predictors.
>
> — Page 96, *Feature Engineering and Selection*, 2019.

Columns are relatively easy to remove from a NumPy array or Pandas `DataFrame`. One approach is to record all columns that have a single unique value, then delete them from the Pandas `DataFrame` by calling the `drop()` function. The complete example is listed below.

```
# delete columns with a single unique value
from pandas import read_csv
# load the dataset
df = read_csv('oil-spill.csv', header=None)
print(df.shape)
# get number of unique values for each column
counts = df.nunique()
# record columns to delete
to_del = [i for i,v in enumerate(counts) if v == 1]
```

```
print(to_del)
# drop useless columns
df.drop(to_del, axis=1, inplace=True)
print(df.shape)
```

Listing 5.7: Example of deleting columns that have a single value.

Running the example first loads the dataset and reports the number of rows and columns. The number of unique values for each column is calculated, and those columns that have a single unique value are identified. In this case, column index 22. The identified columns are then removed from the `DataFrame`, and the number of rows and columns in the `DataFrame` are reported to confirm the change.

```
(937, 50)
[22]
(937, 49)
```

Listing 5.8: Example output from deleting columns that have a single value.

5.5 Consider Columns That Have Very Few Values

In the previous section, we saw that some columns in the example dataset had very few unique values. For example, there were columns that only had 2, 4, and 9 unique values. This might make sense for ordinal or categorical variables. In this case, however, the dataset only contains numerical variables. As such, only having 2, 4, or 9 unique numerical values in a column might be surprising. We can refer to these columns or predictors as near-zero variance predictors, as their variance is not zero, but a very small number close to zero.

> ... near-zero variance predictors or have the potential to have near zero variance during the resampling process. These are predictors that have few unique values (such as two values for binary dummy variables) and occur infrequently in the data.
>
> — Pages 96-97, *Feature Engineering and Selection*, 2019.

These columns may or may not contribute to the skill of a model. We can't assume that they are useless to modeling.

> Although near-zero variance predictors likely contain little valuable predictive information, we may not desire to filter these out.
>
> — Page 97, *Feature Engineering and Selection*, 2019.

Depending on the choice of data preparation and modeling algorithms, variables with very few numerical values can also cause errors or unexpected results. For example, I have seen them cause errors when using power transforms for data preparation and when fitting linear models that assume a *sensible* data probability distribution. To help highlight columns of this type, you can calculate the number of unique values for each variable as a percentage of the total number of rows in the dataset. Let's do this manually using NumPy. The complete example is listed below.

```
# summarize the percentage of unique values for each column using numpy
from numpy import loadtxt
from numpy import unique
# load the dataset
data = loadtxt('oil-spill.csv', delimiter=',')
# summarize the number of unique values in each column
for i in range(data.shape[1]):
  num = len(unique(data[:, i]))
  percentage = float(num) / data.shape[0] * 100
  print('%d, %d, %.1f%%' % (i, num, percentage))
```

Listing 5.9: Example of reporting the variance of each variable.

Running the example reports the column index and the number of unique values for each column, followed by the percentage of unique values out of all rows in the dataset. Here, we can see that some columns have a very low percentage of unique values, such as below 1 percent.

```
0, 238, 25.4%
1, 297, 31.7%
2, 927, 98.9%
3, 933, 99.6%
4, 179, 19.1%
5, 375, 40.0%
6, 820, 87.5%
7, 618, 66.0%
8, 561, 59.9%
9, 57, 6.1%
10, 577, 61.6%
11, 59, 6.3%
12, 73, 7.8%
13, 107, 11.4%
14, 53, 5.7%
15, 91, 9.7%
16, 893, 95.3%
17, 810, 86.4%
18, 170, 18.1%
19, 53, 5.7%
20, 68, 7.3%
21, 9, 1.0%
22, 1, 0.1%
23, 92, 9.8%
24, 9, 1.0%
25, 8, 0.9%
26, 9, 1.0%
27, 308, 32.9%
28, 447, 47.7%
29, 392, 41.8%
30, 107, 11.4%
31, 42, 4.5%
32, 4, 0.4%
33, 45, 4.8%
34, 141, 15.0%
35, 110, 11.7%
36, 3, 0.3%
37, 758, 80.9%
38, 9, 1.0%
```

```
39, 9, 1.0%
40, 388, 41.4%
41, 220, 23.5%
42, 644, 68.7%
43, 649, 69.3%
44, 499, 53.3%
45, 2, 0.2%
46, 937, 100.0%
47, 169, 18.0%
48, 286, 30.5%
49, 2, 0.2%
```

Listing 5.10: Example output from reporting the variance of each variable.

We can update the example to only summarize those variables that have unique values that are less than 1 percent of the number of rows.

```
# summarize the percentage of unique values for each column using numpy
from numpy import loadtxt
from numpy import unique
# load the dataset
data = loadtxt('oil-spill.csv', delimiter=',')
# summarize the number of unique values in each column
for i in range(data.shape[1]):
  num = len(unique(data[:, i]))
  percentage = float(num) / data.shape[0] * 100
  if percentage < 1:
    print('%d, %d, %.1f%%' % (i, num, percentage))
```

Listing 5.11: Example of reporting on columns with low variance.

Running the example, we can see that 11 of the 50 variables have numerical variables that have unique values that are less than 1 percent of the number of rows. This does not mean that these rows and columns should be deleted, but they require further attention. For example:

- Perhaps the unique values can be encoded as ordinal values?
- Perhaps the unique values can be encoded as categorical values?
- Perhaps compare model skill with each variable removed from the dataset?

```
21, 9, 1.0%
22, 1, 0.1%
24, 9, 1.0%
25, 8, 0.9%
26, 9, 1.0%
32, 4, 0.4%
36, 3, 0.3%
38, 9, 1.0%
39, 9, 1.0%
45, 2, 0.2%
49, 2, 0.2%
```

Listing 5.12: Example output from reporting on columns with low variance.

For example, if we wanted to delete all 11 columns with unique values less than 1 percent of rows; the example below demonstrates this.

```
# delete columns where number of unique values is less than 1% of the rows
from pandas import read_csv
# load the dataset
df = read_csv('oil-spill.csv', header=None)
print(df.shape)
# get number of unique values for each column
counts = df.nunique()
# record columns to delete
to_del = [i for i,v in enumerate(counts) if (float(v)/df.shape[0]*100) < 1]
print(to_del)
# drop useless columns
df.drop(to_del, axis=1, inplace=True)
print(df.shape)
```

Listing 5.13: Example of removing columns with low variance.

Running the example first loads the dataset and reports the number of rows and columns. The number of unique values for each column is calculated, and those columns that have a number of unique values less than 1 percent of the rows are identified. In this case, 11 columns. The identified columns are then removed from the `DataFrame`, and the number of rows and columns in the `DataFrame` are reported to confirm the change.

```
(937, 50)
[21, 22, 24, 25, 26, 32, 36, 38, 39, 45, 49]
(937, 39)
```

Listing 5.14: Example output from removing columns with low variance.

5.6 Remove Columns That Have A Low Variance

Another approach to the problem of removing columns with few unique values is to consider the variance of the column. Recall that the variance is a statistic calculated on a variable as the average squared difference of values in the sample from the mean. The variance can be used as a filter for identifying columns to be removed from the dataset. A column that has a single value has a variance of 0.0, and a column that has very few unique values may have a small variance.

The `VarianceThreshold` class from the scikit-learn library supports this as a type of feature selection. An instance of the class can be created and we can specify the `threshold` argument, which defaults to 0.0 to remove columns with a single value. It can then be fit and applied to a dataset by calling the `fit_transform()` function to create a transformed version of the dataset where the columns that have a variance lower than the threshold have been removed automatically.

```
...
# define the transform
transform = VarianceThreshold()
# transform the input data
X_sel = transform.fit_transform(X)
```

Listing 5.15: Example of how to configure and apply the `VarianceThreshold` to data.

We can demonstrate this on the oil spill dataset as follows:

```
# example of applying the variance threshold for feature selection
from pandas import read_csv
from sklearn.feature_selection import VarianceThreshold
# load the dataset
df = read_csv('oil-spill.csv', header=None)
# split data into inputs and outputs
data = df.values
X = data[:, :-1]
y = data[:, -1]
print(X.shape, y.shape)
# define the transform
transform = VarianceThreshold()
# transform the input data
X_sel = transform.fit_transform(X)
print(X_sel.shape)
```

Listing 5.16: Example of removing columns that have a low variance.

Running the example first loads the dataset, then applies the transform to remove all columns with a variance of 0.0. The shape of the dataset is reported before and after the transform, and we can see that the single column where all values are the same has been removed.

```
(937, 49) (937,)
(937, 48)
```

Listing 5.17: Example output from removing columns that have a low variance.

We can expand this example and see what happens when we use different thresholds. We can define a sequence of thresholds from 0.0 to 0.5 with a step size of 0.05, e.g. 0.0, 0.05, 0.1, etc.

```
...
# define thresholds to check
thresholds = arange(0.0, 0.55, 0.05)
```

Listing 5.18: Example of defining variance thresholds to consider.

We can then report the number of features in the transformed dataset for each given threshold.

```
...
# apply transform with each threshold
results = list()
for t in thresholds:
  # define the transform
  transform = VarianceThreshold(threshold=t)
  # transform the input data
  X_sel = transform.fit_transform(X)
  # determine the number of input features
  n_features = X_sel.shape[1]
  print('>Threshold=%.2f, Features=%d' % (t, n_features))
  # store the result
  results.append(n_features)
```

Listing 5.19: Example of evaluating the effect of different variance thresholds.

Finally, we can plot the results. Tying this together, the complete example of comparing variance threshold to the number of selected features is listed below.

```
# explore the effect of the variance thresholds on the number of selected features
from numpy import arange
from pandas import read_csv
from sklearn.feature_selection import VarianceThreshold
from matplotlib import pyplot
# load the dataset
df = read_csv('oil-spill.csv', header=None)
# split data into inputs and outputs
data = df.values
X = data[:, :-1]
y = data[:, -1]
print(X.shape, y.shape)
# define thresholds to check
thresholds = arange(0.0, 0.55, 0.05)
# apply transform with each threshold
results = list()
for t in thresholds:
  # define the transform
  transform = VarianceThreshold(threshold=t)
  # transform the input data
  X_sel = transform.fit_transform(X)
  # determine the number of input features
  n_features = X_sel.shape[1]
  print('>Threshold=%.2f, Features=%d' % (t, n_features))
  # store the result
  results.append(n_features)
# plot the threshold vs the number of selected features
pyplot.plot(thresholds, results)
pyplot.show()
```

Listing 5.20: Example of reviewing the effect of different variance thresholds on the number of features in the transformed dataset.

Running the example first loads the data and confirms that the raw dataset has 49 columns. Next, the `VarianceThreshold` is applied to the raw dataset with values from 0.0 to 0.5 and the number of remaining features after the transform is applied are reported. We can see that the number of features in the dataset quickly drops from 49 in the unchanged data down to 35 with a threshold of 0.15. It later drops to 31 (18 columns deleted) with a threshold of 0.5.

```
(937, 49) (937,)
>Threshold=0.00, Features=48
>Threshold=0.05, Features=37
>Threshold=0.10, Features=36
>Threshold=0.15, Features=35
>Threshold=0.20, Features=35
>Threshold=0.25, Features=35
>Threshold=0.30, Features=35
>Threshold=0.35, Features=35
>Threshold=0.40, Features=35
>Threshold=0.45, Features=33
>Threshold=0.50, Features=31
```

Listing 5.21: Example output from reviewing the effect of different variance thresholds on the

number of features in the transformed dataset.

A line plot is then created showing the relationship between the threshold and the number of features in the transformed dataset. We can see that even with a small threshold between 0.15 and 0.4, that a large number of features (14) are removed immediately.

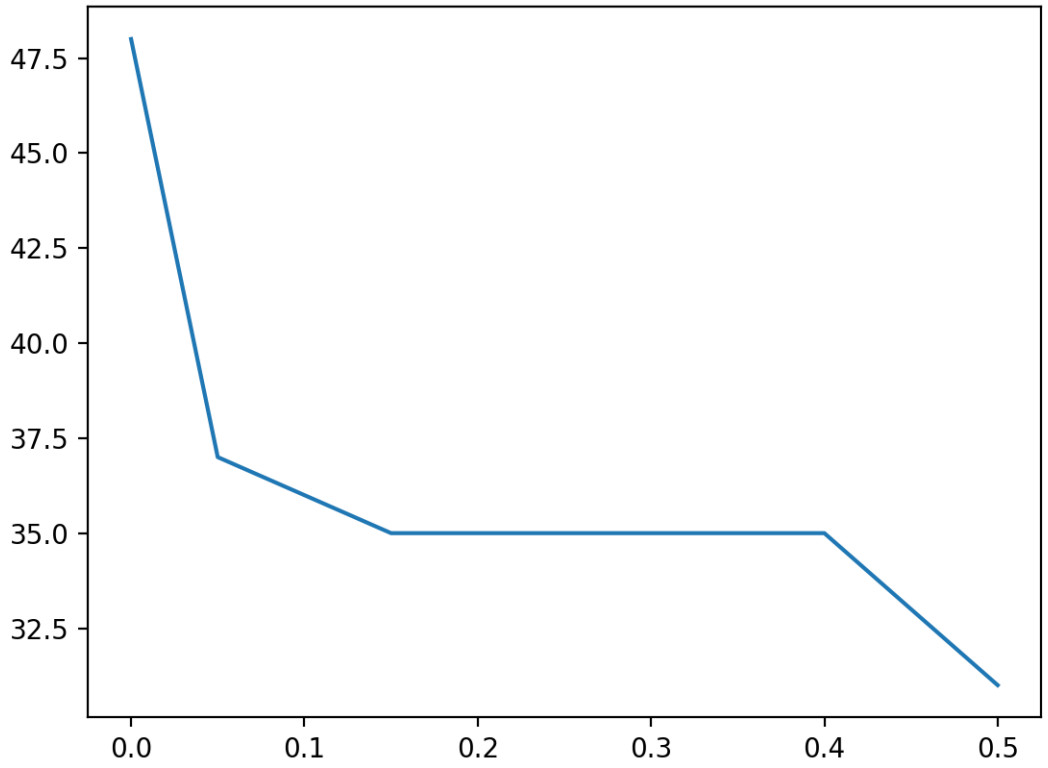

Figure 5.1: Line Plot of Variance Threshold Versus Number of Selected Features.

5.7 Identify Rows That Contain Duplicate Data

Rows that have identical data are could be useless to the modeling process, if not dangerously misleading during model evaluation. Here, a duplicate row is a row where each value in each column for that row appears in identically the same order (same column values) in another row.

> ... if you have used raw data that may have duplicate entries, removing duplicate data will be an important step in ensuring your data can be accurately used.
>
> — Page 173, *Data Wrangling with Python*, 2016.

From a probabilistic perspective, you can think of duplicate data as adjusting the priors for a class label or data distribution. This may help an algorithm like Naive Bayes if you wish to purposefully bias the priors. Typically, this is not the case and machine learning algorithms

will perform better by identifying and removing rows with duplicate data. From an algorithm evaluation perspective, duplicate rows will result in misleading performance. For example, if you are using a train/test split or k-fold cross-validation, then it is possible for a duplicate row or rows to appear in both train and test datasets and any evaluation of the model on these rows will be (or should be) correct. This will result in an optimistically biased estimate of performance on unseen data.

> Data deduplication, also known as duplicate detection, record linkage, record matching, or entity resolution, refers to the process of identifying tuples in one or more relations that refer to the same real-world entity.
>
> — Page 47, *Data Cleaning*, 2019.

If you think this is not the case for your dataset or chosen model, design a controlled experiment to test it. This could be achieved by evaluating model skill with the raw dataset and the dataset with duplicates removed and comparing performance. Another experiment might involve augmenting the dataset with different numbers of randomly selected duplicate examples. The Pandas function `duplicated()` will report whether a given row is duplicated or not. All rows are marked as either `False` to indicate that it is not a duplicate or `True` to indicate that it is a duplicate. If there are duplicates, the first occurrence of the row is marked `False` (by default), as we might expect. The example below checks for duplicates.

```
# locate rows of duplicate data
from pandas import read_csv
# load the dataset
df = read_csv('iris.csv', header=None)
# calculate duplicates
dups = df.duplicated()
# report if there are any duplicates
print(dups.any())
# list all duplicate rows
print(df[dups])
```

Listing 5.22: Example of identifying and reporting duplicate rows.

Running the example first loads the dataset, then calculates row duplicates. First, the presence of any duplicate rows is reported, and in this case, we can see that there are duplicates (`True`). Then all duplicate rows are reported. In this case, we can see that three duplicate rows that were identified are printed.

```
True
      0    1    2    3              4
34   4.9  3.1  1.5  0.1    Iris-setosa
37   4.9  3.1  1.5  0.1    Iris-setosa
142  5.8  2.7  5.1  1.9  Iris-virginica
```

Listing 5.23: Example output from identifying and reporting duplicate rows.

5.8 Delete Rows That Contain Duplicate Data

Rows of duplicate data should probably be deleted from your dataset prior to modeling.

> If your dataset simply has duplicate rows, there is no need to worry about preserving the data; it is already a part of the finished dataset and you can merely remove or drop these rows from your cleaned data.
>
> — Page 186, *Data Wrangling with Python*, 2016.

There are many ways to achieve this, although Pandas provides the `drop_duplicates()` function that achieves exactly this. The example below demonstrates deleting duplicate rows from a dataset.

```
# delete rows of duplicate data from the dataset
from pandas import read_csv
# load the dataset
df = read_csv('iris.csv', header=None)
print(df.shape)
# delete duplicate rows
df.drop_duplicates(inplace=True)
print(df.shape)
```

Listing 5.24: Example of removing duplicate rows.

Running the example first loads the dataset and reports the number of rows and columns. Next, the rows of duplicated data are identified and removed from the `DataFrame`. Then the shape of the `DataFrame` is reported to confirm the change.

```
(150, 5)
(147, 5)
```

Listing 5.25: Example output from removing duplicate rows.

5.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.9.1 Books

- *Data Cleaning*, 2019.
  https://amzn.to/2SARxFG
- *Data Wrangling with Python*, 2016.
  https://amzn.to/35DoLcU
- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn

5.9.2 APIs

- `numpy.unique` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.unique.html

- `pandas.DataFrame.nunique` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.nunique.html
- `pandas.DataFrame.drop` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html
- `pandas.DataFrame.duplicated` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.duplicated.html
- `pandas.DataFrame.drop_duplicates` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop_duplicates.html

5.10 Summary

In this tutorial, you discovered basic data cleaning you should always perform on your dataset. Specifically, you learned:

- How to identify and remove column variables that only have a single value.
- How to identify and consider column variables with very few unique values.
- How to identify and remove rows that contain duplicate observations.

5.10.1 Next

In the next section, we will explore how to identify and remove outliers from data variables.

Chapter 6

Outlier Identification and Removal

When modeling, it is important to clean the data sample to ensure that the observations best represent the problem. Sometimes a dataset can contain extreme values that are outside the range of what is expected and unlike the other data. These are called outliers and often machine learning modeling and model skill in general can be improved by understanding and even removing these outlier values. In this tutorial, you will discover outliers and how to identify and remove them from your machine learning dataset. After completing this tutorial, you will know:

- That an outlier is an unlikely observation in a dataset and may have one of many causes.
- How to use simple univariate statistics like standard deviation and interquartile range to identify and remove outliers from a data sample.
- How to use an outlier detection model to identify and remove rows from a training dataset in order to lift predictive modeling performance.

6.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What are Outliers?
2. Test Dataset
3. Standard Deviation Method
4. Interquartile Range Method
5. Automatic Outlier Detection

6.2 What are Outliers?

An outlier is an observation that is unlike the other observations. They are rare, distinct, or do not fit in some way.

> We will generally define outliers as samples that are exceptionally far from the mainstream of the data.

— Page 33, *Applied Predictive Modeling*, 2013.

Outliers can have many causes, such as:

- Measurement or input error.
- Data corruption.
- True outlier observation.

There is no precise way to define and identify outliers in general because of the specifics of each dataset. Instead, you, or a domain expert, must interpret the raw observations and decide whether a value is an outlier or not.

> Even with a thorough understanding of the data, outliers can be hard to define. [...] Great care should be taken not to hastily remove or change values, especially if the sample size is small.

— Page 33, *Applied Predictive Modeling*, 2013.

Nevertheless, we can use statistical methods to identify observations that appear to be rare or unlikely given the available data.

> Identifying outliers and bad data in your dataset is probably one of the most difficult parts of data cleanup, and it takes time to get right. Even if you have a deep understanding of statistics and how outliers might affect your data, it's always a topic to explore cautiously.

— Page 167, *Data Wrangling with Python*, 2016.

This does not mean that the values identified are outliers and should be removed. But, the tools described in this tutorial can be helpful in shedding light on rare events that may require a second look. A good tip is to consider plotting the identified outlier values, perhaps in the context of non-outlier values to see if there are any systematic relationship or pattern to the outliers. If there is, perhaps they are not outliers and can be explained, or perhaps the outliers themselves can be identified more systematically.

6.3 Test Dataset

Before we look at outlier identification methods, let's define a dataset we can use to test the methods. We will generate a population 10,000 random numbers drawn from a Gaussian distribution with a mean of 50 and a standard deviation of 5. Numbers drawn from a Gaussian distribution will have outliers. That is, by virtue of the distribution itself, there will be a few values that will be a long way from the mean, rare values that we can identify as outliers.

We will use the `randn()` function to generate random Gaussian values with a mean of 0 and a standard deviation of 1, then multiply the results by our own standard deviation and add the mean to shift the values into the preferred range. The pseudorandom number generator is seeded to ensure that we get the same sample of numbers each time the code is run.

```
# generate gaussian data
from numpy.random import seed
from numpy.random import randn
from numpy import mean
from numpy import std
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# summarize
print('mean=%.3f stdv=%.3f' % (mean(data), std(data)))
```

Listing 6.1: Example of a synthetic dataset with outliers.

Running the example generates the sample and then prints the mean and standard deviation. As expected, the values are very close to the expected values.

```
mean=50.049 stdv=4.994
```

Listing 6.2: Example output from summarizing a synthetic dataset with outliers.

6.4 Standard Deviation Method

If we know that the distribution of values in the sample is Gaussian or Gaussian-like, we can use the standard deviation of the sample as a cut-off for identifying outliers. The Gaussian distribution has the property that the standard deviation from the mean can be used to reliably summarize the percentage of values in the sample. For example, within one standard deviation of the mean will cover 68 percent of the data. So, if the mean is 50 and the standard deviation is 5, as in the test dataset above, then all data in the sample between 45 and 55 will account for about 68 percent of the data sample. We can cover more of the data sample if we expand the range as follows:

- 1 Standard Deviation from the Mean: 68 percent.
- 2 Standard Deviations from the Mean: 95 percent.
- 3 Standard Deviations from the Mean: 99.7 percent.

A value that falls outside of 3 standard deviations is part of the distribution, but it is an unlikely or rare event at approximately 1 in 370 samples. Three standard deviations from the mean is a common cut-off in practice for identifying outliers in a Gaussian or Gaussian-like distribution. For smaller samples of data, perhaps a value of 2 standard deviations (95 percent) can be used, and for larger samples, perhaps a value of 4 standard deviations (99.9 percent) can be used.

> Given mu and sigma, a simple way to identify outliers is to compute a z-score for every x_i, which is defined as the number of standard deviations away x_i is from the mean [...] Data values that have a z-score sigma greater than a threshold, for example, of three, are declared to be outliers.

— Page 19, *Data Cleaning*, 2019.

Let's make this concrete with a worked example. Sometimes, the data is standardized first (e.g. to a Z-score with zero mean and unit variance) so that the outlier detection can be performed using standard Z-score cut-off values. This is a convenience and is not required in general, and we will perform the calculations in the original scale of the data here to make things clear. We can calculate the mean and standard deviation of a given sample, then calculate the cut-off for identifying outliers as more than 3 standard deviations from the mean.

```
...
# calculate summary statistics
data_mean, data_std = mean(data), std(data)
# define outliers
cut_off = data_std * 3
lower, upper = data_mean - cut_off, data_mean + cut_off
```

Listing 6.3: Example of estimating the lower and upper bounds of the data.

We can then identify outliers as those examples that fall outside of the defined lower and upper limits.

```
...
# identify outliers
outliers = [x for x in data if x < lower or x > upper]
```

Listing 6.4: Example of identifying outliers using the limits on the data.

Alternately, we can filter out those values from the sample that are not within the defined limits.

```
...
# remove outliers
outliers_removed = [x for x in data if x > lower and x < upper]
```

Listing 6.5: Example of removing outliers from the data.

We can put this all together with our sample dataset prepared in the previous section. The complete example is listed below.

```
# identify outliers with standard deviation
from numpy.random import seed
from numpy.random import randn
from numpy import mean
from numpy import std
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# calculate summary statistics
data_mean, data_std = mean(data), std(data)
# define outliers
cut_off = data_std * 3
lower, upper = data_mean - cut_off, data_mean + cut_off
# identify outliers
outliers = [x for x in data if x < lower or x > upper]
print('Identified outliers: %d' % len(outliers))
# remove outliers
```

```
outliers_removed = [x for x in data if x >= lower and x <= upper]
print('Non-outlier observations: %d' % len(outliers_removed))
```

Listing 6.6: Example of a identifying and removing outliers using the standard deviation.

Running the example will first print the number of identified outliers and then the number of observations that are not outliers, demonstrating how to identify and filter out outliers respectively.

```
Identified outliers: 29
Non-outlier observations: 9971
```

Listing 6.7: Example output from identifying and removing outliers using the standard deviation.

So far we have only talked about univariate data with a Gaussian distribution, e.g. a single variable. You can use the same approach if you have multivariate data, e.g. data with multiple variables, each with a different Gaussian distribution. You can imagine bounds in two dimensions that would define an ellipse if you have two variables. Observations that fall outside of the ellipse would be considered outliers. In three dimensions, this would be an ellipsoid, and so on into higher dimensions. Alternately, if you knew more about the domain, perhaps an outlier may be identified by exceeding the limits on one or a subset of the data dimensions.

6.5 Interquartile Range Method

Not all data is normal or normal enough to treat it as being drawn from a Gaussian distribution. A good statistic for summarizing a non-Gaussian distribution sample of data is the Interquartile Range, or IQR for short. The IQR is calculated as the difference between the 75th and the 25th percentiles of the data and defines the box in a box and whisker plot. Remember that percentiles can be calculated by sorting the observations and selecting values at specific indices. The 50th percentile is the middle value, or the average of the two middle values for an even number of examples. If we had 10,000 samples, then the 50th percentile would be the average of the 5000th and 5001st values.

We refer to the percentiles as quartiles (*quart* meaning 4) because the data is divided into four groups via the 25th, 50th and 75th values. The IQR defines the middle 50 percent of the data, or the body of the data.

> Statistics-based outlier detection techniques assume that the normal data points would appear in high probability regions of a stochastic model, while outliers would occur in the low probability regions of a stochastic model.
>
> — Page 12, *Data Cleaning*, 2019.

The IQR can be used to identify outliers by defining limits on the sample values that are a factor k of the IQR below the 25th percentile or above the 75th percentile. The common value for the factor k is the value 1.5. A factor k of 3 or more can be used to identify values that are extreme outliers or *far outs* when described in the context of box and whisker plots. On a box and whisker plot, these limits are drawn as fences on the whiskers (or the lines) that are drawn from the box. Values that fall outside of these values are drawn as dots. We can calculate the percentiles of a dataset using the `percentile()` NumPy function that takes the dataset and specification of the desired percentile. The IQR can then be calculated as the difference between the 75th and 25th percentiles.

```
...
# calculate interquartile range
q25, q75 = percentile(data, 25), percentile(data, 75)
iqr = q75 - q25
```

Listing 6.8: Example of calculating quartiles on the data.

We can then calculate the cutoff for outliers as 1.5 times the IQR and subtract this cut-off from the 25th percentile and add it to the 75th percentile to give the actual limits on the data.

```
...
# calculate the outlier cutoff
cut_off = iqr * 1.5
lower, upper = q25 - cut_off, q75 + cut_off
```

Listing 6.9: Example of calculating lower and upper bounds using the IQR.

We can then use these limits to identify the outlier values.

```
...
# identify outliers
outliers = [x for x in data if x < lower or x > upper]
```

Listing 6.10: Example of identifying outliers using the limits on the data.

We can also use the limits to filter out the outliers from the dataset.

```
...
# remove outliers
outliers_removed = [x for x in data if x > lower and x < upper]
```

Listing 6.11: Example of removing outliers from the data.

We can tie all of this together and demonstrate the procedure on the test dataset. The complete example is listed below.

```
# identify outliers with interquartile range
from numpy.random import seed
from numpy.random import randn
from numpy import percentile
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# calculate interquartile range
q25, q75 = percentile(data, 25), percentile(data, 75)
iqr = q75 - q25
print('Percentiles: 25th=%.3f, 75th=%.3f, IQR=%.3f' % (q25, q75, iqr))
# calculate the outlier cutoff
cut_off = iqr * 1.5
lower, upper = q25 - cut_off, q75 + cut_off
# identify outliers
outliers = [x for x in data if x < lower or x > upper]
print('Identified outliers: %d' % len(outliers))
# remove outliers
outliers_removed = [x for x in data if x >= lower and x <= upper]
print('Non-outlier observations: %d' % len(outliers_removed))
```

Listing 6.12: Example of a identifying and removing outliers using the IQR.

Running the example first prints the identified 25th and 75th percentiles and the calculated IQR. The number of outliers identified is printed followed by the number of non-outlier observations.

```
Percentiles: 25th=46.685, 75th=53.359, IQR=6.674
Identified outliers: 81
Non-outlier observations: 9919
```

Listing 6.13: Example output from identifying and removing outliers using the IQR.

The approach can be used for multivariate data by calculating the limits on each variable in the dataset in turn, and taking outliers as observations that fall outside of the rectangle or hyper-rectangle.

6.6 Automatic Outlier Detection

In machine learning, an approach to tackling the problem of outlier detection is one-class classification.

> A one-class classifier aims at capturing characteristics of training instances, in order to be able to distinguish between them and potential outliers to appear.
>
> — Page 139, *Learning from Imbalanced Data Sets*, 2018.

A simple approach to identifying outliers is to locate those examples that are far from the other examples in the multi-dimensional feature space. This can work well for feature spaces with low dimensionality (few features), although it can become less reliable as the number of features is increased, referred to as the curse of dimensionality. The local outlier factor, or LOF for short, is a technique that attempts to harness the idea of nearest neighbors for outlier detection. Each example is assigned a scoring of how isolated or how likely it is to be outliers based on the size of its local neighborhood. Those examples with the largest score are more likely to be outliers. The scikit-learn library provides an implementation of this approach in the `LocalOutlierFactor` class.

We can demonstrate the `LocalOutlierFactor` method on a predictive modeling dataset. We will use the Boston housing regression problem that has 13 inputs and one numerical target and requires learning the relationship between suburb characteristics and house prices. You can learn more about the dataset here:

- Boston Housing Dataset (`housing.csv`).[1]
- Boston Housing Dataset Description (`housing.names`).[2]

Looking in the dataset, you should see that all variables are numeric.

```
0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,1,296.0,15.30,396.90,4.98,24.00
0.02731,0.00,7.070,0,0.4690,6.4210,78.90,4.9671,2,242.0,17.80,396.90,9.14,21.60
0.02729,0.00,7.070,0,0.4690,7.1850,61.10,4.9671,2,242.0,17.80,392.83,4.03,34.70
0.03237,0.00,2.180,0,0.4580,6.9980,45.80,6.0622,3,222.0,18.70,394.63,2.94,33.40
```

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv
[2] https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.names

Chapter 17

How to Scale Numerical Data

Many machine learning algorithms perform better when numerical input variables are scaled to a standard range. This includes algorithms that use a weighted sum of the input, like linear regression, and algorithms that use distance measures, like k-nearest neighbors. The two most popular techniques for scaling numerical data prior to modeling are normalization and standardization. Normalization scales each input variable separately to the range 0-1, which is the range for floating-point values where we have the most precision. Standardization scales each input variable separately by subtracting the mean (called centering) and dividing by the standard deviation to shift the distribution to have a mean of zero and a standard deviation of one. In this tutorial, you will discover how to use scaler transforms to standardize and normalize numerical input variables for classification and regression. After completing this tutorial, you will know:

- Data scaling is a recommended pre-processing step when working with many machine learning algorithms.
- Data scaling can be achieved by normalizing or standardizing real-valued input and output variables.
- How to apply standardization and normalization to improve the performance of predictive modeling algorithms.

Let's get started.

17.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. The Scale of Your Data Matters
2. Numerical Data Scaling Methods
3. Diabetes Dataset
4. `MinMaxScaler` Transform
5. `StandardScaler` Transform
6. Common Questions

17.2 The Scale of Your Data Matters

Machine learning models learn a mapping from input variables to an output variable. As such, the scale and distribution of the data drawn from the domain may be different for each variable. Input variables may have different units (e.g. feet, kilometers, and hours) that, in turn, may mean the variables have different scales. Differences in the scales across input variables may increase the difficulty of the problem being modeled. An example of this is that large input values (e.g. a spread of hundreds or thousands of units) can result in a model that learns large weight values. A model with large weight values is often unstable, meaning that it may suffer from poor performance during learning and sensitivity to input values resulting in higher generalization error.

> One of the most common forms of pre-processing consists of a simple linear rescaling of the input variables.
>
> — Page 298, *Neural Networks for Pattern Recognition*, 1995.

This difference in scale for input variables does not affect all machine learning algorithms. For example, algorithms that fit a model that use a weighted sum of input variables are affected, such as linear regression, logistic regression, and artificial neural networks (deep learning).

> For example, when the distance or dot products between predictors are used (such as K-nearest neighbors or support vector machines) or when the variables are required to be a common scale in order to apply a penalty, a standardization procedure is essential.
>
> — Page 124, *Feature Engineering and Selection*, 2019.

Also, algorithms that use distance measures between examples are affected, such as k-nearest neighbors and support vector machines. There are also algorithms that are unaffected by the scale of numerical input variables, most notably decision trees and ensembles of trees, like random forest.

> Different attributes are measured on different scales, so if the Euclidean distance formula were used directly, the effect of some attributes might be completely dwarfed by others that had larger scales of measurement. Consequently, it is usual to normalize all attribute values ...
>
> — Page 145, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

It can also be a good idea to scale the target variable for regression predictive modeling problems to make the problem easier to learn, most notably in the case of neural network models. A target variable with a large spread of values, in turn, may result in large error gradient values causing weight values to change dramatically, making the learning process unstable. Scaling input and output variables is a critical step in using neural network models.

> In practice, it is nearly always advantageous to apply pre-processing transformations to the input data before it is presented to a network. Similarly, the outputs of the network are often post-processed to give the required output values.
>
> — Page 296, *Neural Networks for Pattern Recognition*, 1995.

17.3 Numerical Data Scaling Methods

Both normalization and standardization can be achieved using the scikit-learn library. Let's take a closer look at each in turn.

17.3.1 Data Normalization

Normalization is a rescaling of the data from the original range so that all values are within the new range of 0 and 1. Normalization requires that you know or are able to accurately estimate the minimum and maximum observable values. You may be able to estimate these values from your available data.

> Attributes are often normalized to lie in a fixed range - usually from zero to one- by dividing all values by the maximum value encountered or by subtracting the minimum value and dividing by the range between the maximum and minimum values.
>
> — Page 61, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

A value is normalized as follows:

$$y = \frac{x - \min}{\max - \min} \tag{17.1}$$

Where the minimum and maximum values pertain to the value x being normalized. For example, for a dataset, we could guesstimate the min and max observable values as 30 and -10. We can then normalize any value, like 18.8, as follows:

$$\begin{aligned} y &= \frac{x - \min}{\max - \min} \\ &= \frac{18.8 - -10}{30 - -10} \\ &= \frac{28.8}{40} \\ &= 0.72 \end{aligned} \tag{17.2}$$

You can see that if an x value is provided that is outside the bounds of the minimum and maximum values, the resulting value will not be in the range of 0 and 1. You could check for these observations prior to making predictions and either remove them from the dataset or limit them to the pre-defined maximum or minimum values. You can normalize your dataset using the scikit-learn object `MinMaxScaler`. Good practice usage with the `MinMaxScaler` and other scaling techniques is as follows:

- **Fit the scaler using available training data**. For normalization, this means the training data will be used to estimate the minimum and maximum observable values. This is done by calling the `fit()` function.
- **Apply the scale to training data**. This means you can use the normalized data to train your model. This is done by calling the `transform()` function.

- **Apply the scale to data going forward**. This means you can prepare new data in the future on which you want to make predictions.

The default scale for the `MinMaxScaler` is to rescale variables into the range [0,1], although a preferred scale can be specified via the `feature_range` argument as a tuple containing the min and the max for all variables.

```
...
# create scaler
scaler = MinMaxScaler(feature_range=(0,1))
```

Listing 17.1: Example of defining a `MinMaxScaler` instance.

If needed, the transform can be inverted. This is useful for converting predictions back into their original scale for reporting or plotting. This can be done by calling the `inverse_transform()` function. We can demonstrate the usage of this class by converting two variables to a range 0-to-1, the default range for normalization. The first variable has values between about 4 and 100, the second has values between about 0.1 and 0.001. The complete example is listed below.

```
# example of a normalization
from numpy import asarray
from sklearn.preprocessing import MinMaxScaler
# define data
data = asarray([[100, 0.001],
        [8, 0.05],
        [50, 0.005],
        [88, 0.07],
        [4, 0.1]])
print(data)
# define min max scaler
scaler = MinMaxScaler()
# transform data
scaled = scaler.fit_transform(data)
print(scaled)
```

Listing 17.2: Example of normalizing values in a dataset.

Running the example first reports the raw dataset, showing 2 columns with 4 rows. The values are in scientific notation which can be hard to read if you're not used to it. Next, the scaler is defined, fit on the whole dataset and then used to create a transformed version of the dataset with each column normalized independently. We can see that the largest raw value for each column now has the value 1.0 and the smallest value for each column now has the value 0.0.

```
[[1.0e+02 1.0e-03]
 [8.0e+00 5.0e-02]
 [5.0e+01 5.0e-03]
 [8.8e+01 7.0e-02]
 [4.0e+00 1.0e-01]]
[[1.         0.        ]
 [0.04166667 0.49494949]
 [0.47916667 0.04040404]
 [0.875      0.6969697 ]
 [0.         1.        ]]
```

Listing 17.3: Example output from normalizing values in a dataset.

Now that we are familiar with normalization, let's take a closer look at standardization.

17.3.2 Data Standardization

Standardizing a dataset involves rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1. This can be thought of as subtracting the mean value or centering the data. Like normalization, standardization can be useful, and even required in some machine learning algorithms when your data has input values with differing scales. Standardization assumes that your observations fit a Gaussian distribution (bell curve) with a well-behaved mean and standard deviation. You can still standardize your data if this expectation is not met, but you may not get reliable results.

> Another [...] technique is to calculate the statistical mean and standard deviation of the attribute values, subtract the mean from each value, and divide the result by the standard deviation. This process is called standardizing a statistical variable and results in a set of values whose mean is zero and standard deviation is one.
>
> — Page 61, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

Standardization requires that you know or are able to accurately estimate the mean and standard deviation of observable values. You may be able to estimate these values from your training data, not the entire dataset.

> ... it is emphasized that the statistics required for the transformation (e.g., the mean) are estimated from the training set and are applied to all data sets (e.g., the test set or new samples).
>
> — Page 124, *Feature Engineering and Selection*, 2019.

Subtracting the mean from the data is called centering, whereas dividing by the standard deviation is called scaling. As such, the method is sometimes called *center scaling*.

> The most straightforward and common data transformation is to center scale the predictor variables. To center a predictor variable, the average predictor value is subtracted from all the values. As a result of centering, the predictor has a zero mean. Similarly, to scale the data, each value of the predictor variable is divided by its standard deviation. Scaling the data coerce the values to have a common standard deviation of one.
>
> — Page 30, *Applied Predictive Modeling*, 2013.

A value is standardized as follows:

$$y = \frac{x - \text{mean}}{\text{standard\_deviation}} \tag{17.3}$$

Where the mean is calculated as:

$$\text{mean} = \frac{1}{N} \times \sum_{i=1}^{N} x_i \tag{17.4}$$

And the standard_deviation is calculated as:

$$\text{standard\_deviation} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \text{mean})^2}{N-1}} \tag{17.5}$$

We can guesstimate a mean of 10.0 and a standard deviation of about 5.0. Using these values, we can standardize the first value of 20.7 as follows:

$$\begin{aligned} y &= \frac{x - \text{mean}}{\text{standard\_deviation}} \\ &= \frac{20.7 - 10}{5} \\ &= \frac{10.7}{5} \\ &= 2.14 \end{aligned} \tag{17.6}$$

The mean and standard deviation estimates of a dataset can be more robust to new data than the minimum and maximum. You can standardize your dataset using the scikit-learn object `StandardScaler`. We can demonstrate the usage of this class by converting two variables defined in the previous section. We will use the default configuration that will both center and scale the values in each column, e.g. full standardization. The complete example is listed below.

```
# example of a standardization
from numpy import asarray
from sklearn.preprocessing import StandardScaler
# define data
data = asarray([[100, 0.001],
        [8, 0.05],
        [50, 0.005],
        [88, 0.07],
        [4, 0.1]])
print(data)
# define standard scaler
scaler = StandardScaler()
# transform data
scaled = scaler.fit_transform(data)
print(scaled)
```

Listing 17.4: Example of standardizing values in a dataset.

Running the example first reports the raw dataset, showing 2 columns with 4 rows as before. Next, the scaler is defined, fit on the whole dataset and then used to create a transformed version of the dataset with each column standardized independently. We can see that the mean value in each column is assigned a value of 0.0 if present and the values are centered around 0.0 with values both positive and negative.

```
[[1.0e+02 1.0e-03]
 [8.0e+00 5.0e-02]
 [5.0e+01 5.0e-03]
 [8.8e+01 7.0e-02]
 [4.0e+00 1.0e-01]]
[[ 1.26398112 -1.16389967]
 [-1.06174414 0.12639634]
 [ 0.        -1.05856939]
```

```
 [ 0.96062565 0.65304778]
 [-1.16286263 1.44302493]]
```

Listing 17.5: Example output from standardizing values in a dataset.

Next, we can introduce a real dataset that provides the basis for applying normalization and standardization transforms as a part of modeling.

17.4 Diabetes Dataset

In this tutorial we will use the diabetes dataset. This dataset classifies patients data as either an onset of diabetes within five years or not and was introduced in Chapter 7. First, let's load and summarize the dataset. The complete example is listed below.

```
# load and summarize the diabetes dataset
from pandas import read_csv
from matplotlib import pyplot
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the shape of the dataset
print(dataset.shape)
# summarize each variable
print(dataset.describe())
# histograms of the variables
fig = dataset.hist(xlabelsize=4, ylabelsize=4)
[x.title.set_size(4) for x in fig.ravel()]
# show the plot
pyplot.show()
```

Listing 17.6: Example of loading and summarizing the diabetes dataset.

Running the example first summarizes the shape of the loaded dataset. This confirms the 7 input variables, one output variable, and 768 rows of data. A statistical summary of the input variables is provided show that each variable has a very different scale. This makes it a good dataset for exploring data scaling methods.

```
(768, 9)
                0           1           2  ...           6           7           8
count  768.000000  768.000000  768.000000  ...  768.000000  768.000000  768.000000
mean     3.845052  120.894531   69.105469  ...    0.471876   33.240885    0.348958
std      3.369578   31.972618   19.355807  ...    0.331329   11.760232    0.476951
min      0.000000    0.000000    0.000000  ...    0.078000   21.000000    0.000000
25%      1.000000   99.000000   62.000000  ...    0.243750   24.000000    0.000000
50%      3.000000  117.000000   72.000000  ...    0.372500   29.000000    0.000000
75%      6.000000  140.250000   80.000000  ...    0.626250   41.000000    1.000000
max     17.000000  199.000000  122.000000  ...    2.420000   81.000000    1.000000
```

Listing 17.7: Example output from summarizing the variables from the diabetes dataset.

Finally, a histogram is created for each input variable. The plots confirm the differing scale for each input variable and show that the variables have differing scales.

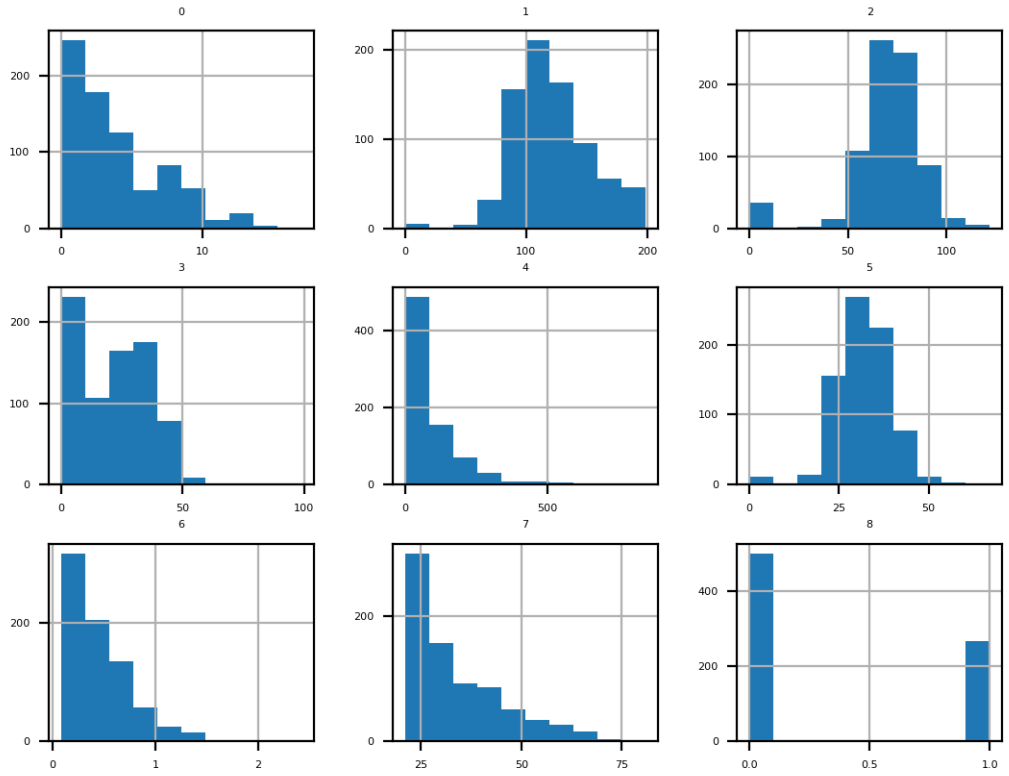

Figure 17.1: Histogram Plots of Input Variables for the Diabetes Binary Classification Dataset.

Next, let's fit and evaluate a machine learning model on the raw dataset. We will use a k-nearest neighbor algorithm with default hyperparameters and evaluate it using repeated stratified k-fold cross-validation. The complete example is listed below.

```
# evaluate knn on the raw diabetes dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import LabelEncoder
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
data = dataset.values
# separate into input and output columns
X, y = data[:, :-1], data[:, -1]
# ensure inputs are floats and output is an integer label
X = X.astype('float32')
y = LabelEncoder().fit_transform(y.astype('str'))
# define and configure the model
model = KNeighborsClassifier()
# evaluate the model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
```

```
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report model performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 17.8: Example of evaluating model performance on the diabetes dataset.

Running the example evaluates a KNN model on the raw diabetes dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm, the evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average performance.

In this case we can see that the model achieved a mean classification accuracy of about 71.7 percent, showing that it has skill (better than 65 percent) and is in the ball-park of good performance (77 percent).

```
Accuracy: 0.717 (0.040)
```

Listing 17.9: Example output from evaluating model performance on the diabetes dataset.

Next, let's explore a scaling transform of the dataset.

17.5 MinMaxScaler Transform

We can apply the `MinMaxScaler` to the diabetes dataset directly to normalize the input variables. We will use the default configuration and scale values to the range 0 and 1. First, a `MinMaxScaler` instance is defined with default hyperparameters. Once defined, we can call the `fit_transform()` function and pass it to our dataset to create a transformed version of our dataset.

```
...
# perform a robust scaler transform of the dataset
trans = MinMaxScaler()
data = trans.fit_transform(data)
```

Listing 17.10: Example of transforming a dataset with the `MinMaxScaler`.

Let's try it on our diabetes dataset. The complete example of creating a `MinMaxScaler` transform of the diabetes dataset and plotting histograms of the result is listed below.

```
# visualize a minmax scaler transform of the diabetes dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import MinMaxScaler
from matplotlib import pyplot
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a robust scaler transform of the dataset
trans = MinMaxScaler()
data = trans.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# summarize
print(dataset.describe())
```

```
# histograms of the variables
fig = dataset.hist(xlabelsize=4, ylabelsize=4)
[x.title.set_size(4) for x in fig.ravel()]
# show the plot
pyplot.show()
```

Listing 17.11: Example of reviewing the data after a MinMaxScaler transform.

Running the example first reports a summary of each input variable. We can see that the distributions have been adjusted and that the minimum and maximum values for each variable are now a crisp 0.0 and 1.0 respectively.

```
                0           1           2  ...           5           6           7
count  768.000000  768.000000  768.000000  ...  768.000000  768.000000  768.000000
mean     0.226180    0.607510    0.566438  ...    0.476790    0.168179    0.204015
std      0.198210    0.160666    0.158654  ...    0.117499    0.141473    0.196004
min      0.000000    0.000000    0.000000  ...    0.000000    0.000000    0.000000
25%      0.058824    0.497487    0.508197  ...    0.406855    0.070773    0.050000
50%      0.176471    0.587940    0.590164  ...    0.476900    0.125747    0.133333
75%      0.352941    0.704774    0.655738  ...    0.545455    0.234095    0.333333
max      1.000000    1.000000    1.000000  ...    1.000000    1.000000    1.000000
```

Listing 17.12: Example output from summarizing the variables from the diabetes dataset after a MinMaxScaler transform.

Histogram plots of the variables are created, although the distributions don't look much different from their original distributions seen in the previous section. We can confirm that the minimum and maximum values are not zero and one respectively, as we expected.

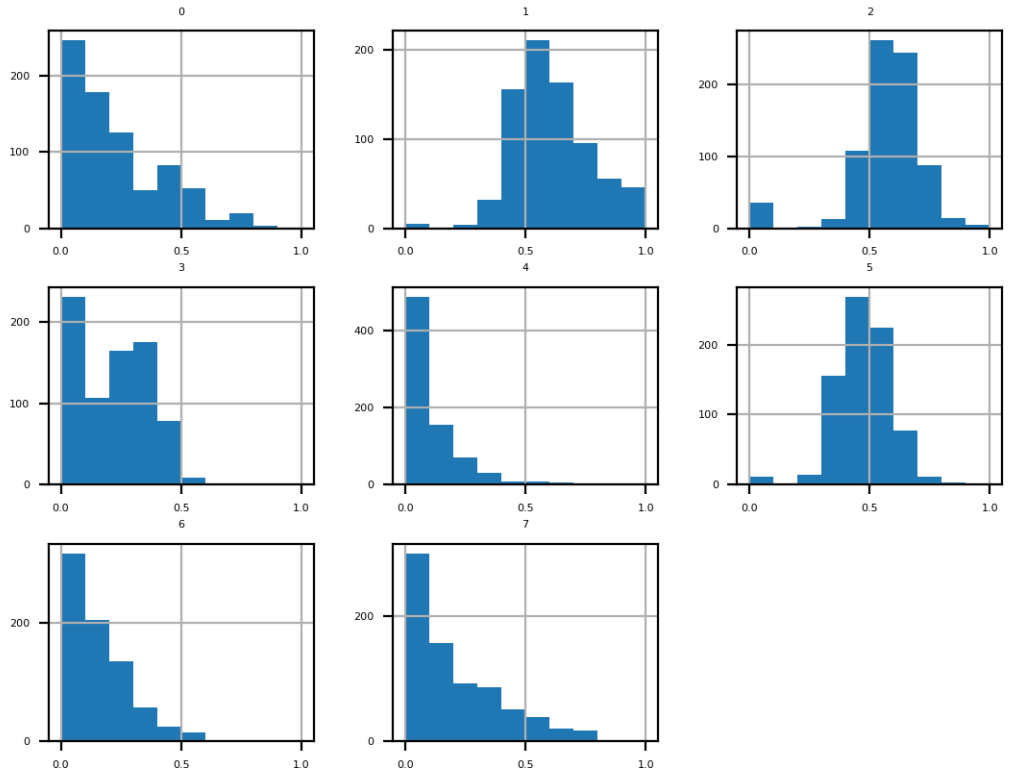

Figure 17.2: Histogram Plots of MinMaxScaler Transformed Input Variables for the Diabetes Dataset.

Next, let's evaluate the same KNN model as the previous section, but in this case, on a MinMaxScaler transform of the dataset. The complete example is listed below.

```
# evaluate knn on the diabetes dataset with minmax scaler transform
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
data = dataset.values
# separate into input and output columns
X, y = data[:, :-1], data[:, -1]
# ensure inputs are floats and output is an integer label
X = X.astype('float32')
y = LabelEncoder().fit_transform(y.astype('str'))
# define the pipeline
trans = MinMaxScaler()
```

```
model = KNeighborsClassifier()
pipeline = Pipeline(steps=[('t', trans), ('m', model)])
# evaluate the pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report pipeline performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 17.13: Example of evaluating model performance after a MinMaxScaler transform.

Running the example, we can see that the MinMaxScaler transform results in a lift in performance from 71.7 percent accuracy without the transform to about 73.9 percent with the transform.

Note: Your specific results may vary given the stochastic nature of the learning algorithm, the evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average performance.

```
Accuracy: 0.739 (0.053)
```

Listing 17.14: Example output from evaluating model performance after a MinMaxScaler transform.

Next, let's explore the effect of standardizing the input variables.

17.6 StandardScaler Transform

We can apply the StandardScaler to the diabetes dataset directly to standardize the input variables. We will use the default configuration and scale values to subtract the mean to center them on 0.0 and divide by the standard deviation to give the standard deviation of 1.0. First, a StandardScaler instance is defined with default hyperparameters. Once defined, we can call the fit_transform() function and pass it to our dataset to create a transformed version of our dataset.

```
...
# perform a robust scaler transform of the dataset
trans = StandardScaler()
data = trans.fit_transform(data)
```

Listing 17.15: Example of transforming a dataset with the StandardScaler.

Let's try it on our diabetes dataset. The complete example of creating a StandardScaler transform of the diabetes dataset and plotting histograms of the results is listed below.

```
# visualize a standard scaler transform of the diabetes dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import StandardScaler
from matplotlib import pyplot
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
```

```
# perform a robust scaler transform of the dataset
trans = StandardScaler()
data = trans.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# summarize
print(dataset.describe())
# histograms of the variables
fig = dataset.hist(xlabelsize=4, ylabelsize=4)
[x.title.set_size(4) for x in fig.ravel()]
# show the plot
pyplot.show()
```

Listing 17.16: Example of reviewing the data after a `StandardScaler` transform.

Running the example first reports a summary of each input variable. We can see that the distributions have been adjusted and that the mean is a very small number close to zero and the standard deviation is very close to 1.0 for each variable.

```
                0             1  ...             6             7
count  7.680000e+02  7.680000e+02  ...  7.680000e+02  7.680000e+02
mean   2.544261e-17  3.614007e-18  ...  2.398978e-16  1.857600e-16
std    1.000652e+00  1.000652e+00  ...  1.000652e+00  1.000652e+00
min   -1.141852e+00 -3.783654e+00  ... -1.189553e+00 -1.041549e+00
25%   -8.448851e-01 -6.852363e-01  ... -6.889685e-01 -7.862862e-01
50%   -2.509521e-01 -1.218877e-01  ... -3.001282e-01 -3.608474e-01
75%    6.399473e-01  6.057709e-01  ...  4.662269e-01  6.602056e-01
max    3.906578e+00  2.444478e+00  ...  5.883565e+00  4.063716e+00
```

Listing 17.17: Example output from summarizing the variables from the diabetes dataset after a `StandardScaler` transform.

Histogram plots of the variables are created, although the distributions don't look much different from their original distributions seen in the previous section other than their scale on the x-axis. We can see that the center of mass for each distribution is centered on zero, which is more obvious for some variables than others.

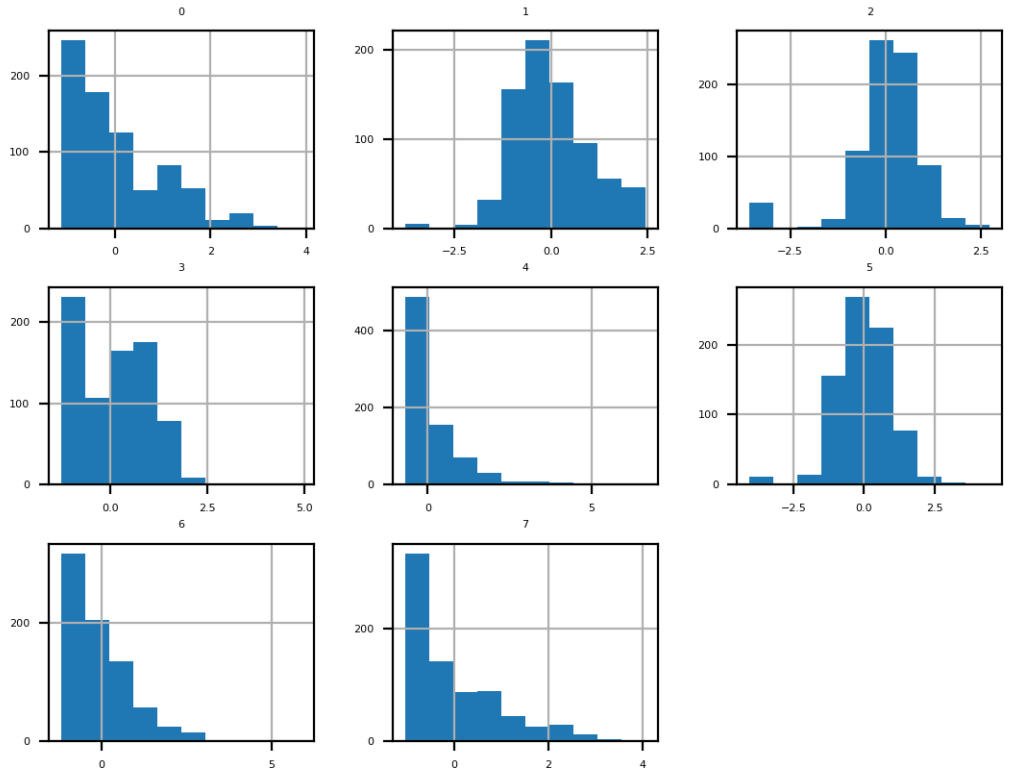

Figure 17.3: Histogram Plots of StandardScaler Transformed Input Variables for the Diabetes Dataset.

Next, let's evaluate the same KNN model as the previous section, but in this case, on a StandardScaler transform of the dataset. The complete example is listed below.

```
# evaluate knn on the diabetes dataset with standard scaler transform
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
data = dataset.values
# separate into input and output columns
X, y = data[:, :-1], data[:, -1]
# ensure inputs are floats and output is an integer label
X = X.astype('float32')
y = LabelEncoder().fit_transform(y.astype('str'))
# define the pipeline
trans = StandardScaler()
```

```
model = KNeighborsClassifier()
pipeline = Pipeline(steps=[('t', trans), ('m', model)])
# evaluate the pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report pipeline performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 17.18: Example of evaluating model performance after a `StandardScaler` transform.

Running the example, we can see that the `StandardScaler` transform results in a lift in performance from 71.7 percent accuracy without the transform to about 74.1 percent with the transform, slightly higher than the result using the `MinMaxScaler` that achieved 73.9 percent.

Note: Your specific results may vary given the stochastic nature of the learning algorithm, the evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average performance.

```
Accuracy: 0.741 (0.050)
```

Listing 17.19: Example output from evaluating model performance after a `StandardScaler` transform.

17.7 Common Questions

This section lists some common questions and answers when scaling numerical data.

Q. Should I Normalize or Standardize?

Whether input variables require scaling depends on the specifics of your problem and of each variable. You may have a sequence of quantities as inputs, such as prices or temperatures. If the distribution of the quantity is normal, then it should be standardized, otherwise, the data should be normalized. This applies if the range of quantity values is large (10s, 100s, etc.) or small (0.01, 0.0001).

> These manipulations are generally used to improve the numerical stability of some calculations. Some models [...] benefit from the predictors being on a common scale.
>
> — Pages 30-31, *Applied Predictive Modeling*, 2013.

If the quantity values are small (near 0-1) and the distribution is limited (e.g. standard deviation near 1), then perhaps you can get away with no scaling of the data. Predictive modeling problems can be complex, and it may not be clear how to best scale input data. If in doubt, normalize the input sequence. If you have the resources, explore modeling with the raw data, standardized data, and normalized data and see if there is a beneficial difference in the performance of the resulting model.

> If the input variables are combined linearly, as in an MLP [Multilayer Perceptron], then it is rarely strictly necessary to standardize the inputs, at least in theory. [...] However, there are a variety of practical reasons why standardizing the inputs can make training faster and reduce the chances of getting stuck in local optima.
>
> — *Should I normalize/standardize/rescale the data? Neural Nets FAQ.*

Chapter 19

How to Encode Categorical Data

Machine learning models require all input and output variables to be numeric. This means that if your data contains categorical data, you must encode it to numbers before you can fit and evaluate a model. The two most popular techniques are an Ordinal encoding and a One Hot encoding. In this tutorial, you will discover how to use encoding schemes for categorical machine learning data. After completing this tutorial, you will know:

- Encoding is a required pre-processing step when working with categorical data for machine learning algorithms.
- How to use ordinal encoding for categorical variables that have a natural rank ordering.
- How to use one hot encoding for categorical variables that do not have a natural rank ordering.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Nominal and Ordinal Variables
2. Encoding Categorical Data
3. Breast Cancer Dataset
4. `OrdinalEncoder` Transform
5. `OneHotEncoder` Transform
6. Common Questions

19.2 Nominal and Ordinal Variables

Numerical data, as its name suggests, involves features that are only composed of numbers, such as integers or floating-point values. Categorical data are variables that contain label values rather than numeric values. The number of possible values is often limited to a fixed set. Categorical variables are often called nominal. Some examples include:

- A *pet* variable with the values: *dog* and *cat*.
- A *color* variable with the values: *red*, *green*, and *blue*.
- A *place* variable with the values: *first*, *second*, and *third*.

Each value represents a different category. Some categories may have a natural relationship to each other, such as a natural ordering. The *place* variable above does have a natural ordering of values. This type of categorical variable is called an ordinal variable because the values can be ordered or ranked. A numerical variable can be converted to an ordinal variable by dividing the range of the numerical variable into bins and assigning values to each bin. For example, a numerical variable between 1 and 10 can be divided into an ordinal variable with 5 labels with an ordinal relationship: 1-2, 3-4, 5-6, 7-8, 9-10. This is called discretization.

- **Nominal Variable**. Variable comprises a finite set of discrete values with no rank-order relationship between values.
- **Ordinal Variable**. Variable comprises a finite set of discrete values with a ranked ordering between values.

Some algorithms can work with categorical data directly. For example, a decision tree can be learned directly from categorical data with no data transform required (this depends on the specific implementation). Many machine learning algorithms cannot operate on label data directly. They require all input variables and output variables to be numeric. In general, this is mostly a constraint of the efficient implementation of machine learning algorithms rather than hard limitations on the algorithms themselves.

Some implementations of machine learning algorithms require all data to be numerical. For example, scikit-learn has this requirement. This means that categorical data must be converted to a numerical form. If the categorical variable is an output variable, you may also want to convert predictions by the model back into a categorical form in order to present them or use them in some application.

19.3 Encoding Categorical Data

There are three common approaches for converting ordinal and categorical variables to numerical values. They are:

- Ordinal Encoding
- One Hot Encoding
- Dummy Variable Encoding

Let's take a closer look at each in turn.

19.3.1 Ordinal Encoding

In ordinal encoding, each unique category value is assigned an integer value. For example, *red* is 1, *green* is 2, and *blue* is 3. This is called an ordinal encoding or an integer encoding and is easily reversible. Often, integer values starting at zero are used. For some variables, an ordinal encoding may be enough. The integer values have a natural ordered relationship between each other and machine learning algorithms may be able to understand and harness this relationship.

An integer ordinal encoding is a natural encoding for ordinal variables. For categorical variables, it imposes an ordinal relationship where no such relationship may exist. This can cause problems and a one hot encoding may be used instead. This ordinal encoding transform is available in the scikit-learn Python machine learning library via the `OrdinalEncoder` class. By default, it will assign integers to labels in the order that is observed in the data. If a specific order is desired, it can be specified via the `categories` argument as a list with the rank order of all expected labels.

We can demonstrate the usage of this class by converting colors categories *red*, *green* and *blue* into integers. First the categories are sorted then numbers are applied. For strings, this means the labels are sorted alphabetically and that *blue=0*, *green=1*, and *red=2*. The complete example is listed below.

```
# example of a ordinal encoding
from numpy import asarray
from sklearn.preprocessing import OrdinalEncoder
# define data
data = asarray([['red'], ['green'], ['blue']])
print(data)
# define ordinal encoding
encoder = OrdinalEncoder()
# transform data
result = encoder.fit_transform(data)
print(result)
```

Listing 19.1: Example of demonstrating an ordinal encoding of color categories.

Running the example first reports the 3 rows of label data, then the ordinal encoding. We can see that the numbers are assigned to the labels as we expected.

```
[['red']
 ['green']
 ['blue']]
[[2.]
 [1.]
 [0.]]
```

Listing 19.2: Example output from demonstrating an ordinal encoding of color categories.

This `OrdinalEncoder` class is intended for input variables that are organized into rows and columns, e.g. a matrix. If a categorical target variable needs to be encoded for a classification predictive modeling problem, then the `LabelEncoder` class can be used. It does the same thing as the `OrdinalEncoder`, although it expects a one-dimensional input for the single target variable.

19.3.2 One Hot Encoding

For categorical variables where no ordinal relationship exists, the integer encoding may not be enough or even misleading to the model. Forcing an ordinal relationship via an ordinal encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results (predictions halfway between categories). In this case, a one hot encoding can be applied to the ordinal representation. This is where the integer encoded variable is removed and one new binary variable is added for each unique integer value in the variable.

> Each bit represents a possible category. If the variable cannot belong to multiple categories at once, then only one bit in the group can be "on". This is called one-hot encoding ...
>
> — Page 78, *Feature Engineering for Machine Learning*, 2018.

In the *color* variable example, there are three categories, and, therefore, three binary variables are needed. A 1 value is placed in the binary variable for the color and 0 values for the other colors. This one hot encoding transform is available in the scikit-learn Python machine learning library via the `OneHotEncoder` class. We can demonstrate the usage of the `OneHotEncoder` on the color categories. First the categories are sorted, in this case alphabetically because they are strings, then binary variables are created for each category in turn. This means blue will be represented as [1, 0, 0] with a 1 in for the first binary variable, then *green*, then finally *red*. The complete example is listed below.

```
# example of a one hot encoding
from numpy import asarray
from sklearn.preprocessing import OneHotEncoder
# define data
data = asarray([['red'], ['green'], ['blue']])
print(data)
# define one hot encoding
encoder = OneHotEncoder(sparse=False)
# transform data
onehot = encoder.fit_transform(data)
print(onehot)
```

Listing 19.3: Example of demonstrating a one hot encoding of color categories.

Running the example first lists the three rows of label data, then the one hot encoding matching our expectation of 3 binary variables in the order *blue*, *green* and *red*.

```
[['red']
 ['green']
 ['blue']]
[[0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]]
```

Listing 19.4: Example output from demonstrating an one hot encoding of color categories.

If you know all of the labels to be expected in the data, they can be specified via the `categories` argument as a list. The encoder is fit on the training dataset, which likely contains

at least one example of all expected labels for each categorical variable if you do not specify the list of labels. If new data contains categories not seen in the training dataset, the `handle_unknown` argument can be set to `'ignore'` to not raise an error, which will result in a zero value for each label.

19.3.3 Dummy Variable Encoding

The one hot encoding creates one binary variable for each category. The problem is that this representation includes redundancy. For example, if we know that [1, 0, 0] represents *blue* and [0, 1, 0] represents *green* we don't need another binary variable to represent *red*, instead we could use 0 values alone, e.g. [0, 0]. This is called a dummy variable encoding, and always represents C categories with $C - 1$ binary variables.

> When there are C possible values of the predictor and only $C - 1$ dummy variables are used, the matrix inverse can be computed and the contrast method is said to be a full rank parameterization
>
> — Page 95, *Feature Engineering and Selection*, 2019.

In addition to being slightly less redundant, a dummy variable representation is required for some models. For example, in the case of a linear regression model (and other regression models that have a bias term), a one hot encoding will cause the matrix of input data to become singular, meaning it cannot be inverted and the linear regression coefficients cannot be calculated using linear algebra. For these types of models a dummy variable encoding must be used instead.

> If the model includes an intercept and contains dummy variables [...], then the [...] columns would add up (row-wise) to the intercept and this linear combination would prevent the matrix inverse from being computed (as it is singular).
>
> — Page 95, *Feature Engineering and Selection*, 2019.

We rarely encounter this problem in practice when evaluating machine learning algorithms, unless we are using linear regression of course.

> ... there are occasions when a complete set of dummy variables is useful. For example, the splits in a tree-based model are more interpretable when the dummy variables encode all the information for that predictor. We recommend using the full set of dummy variables when working with tree-based models.
>
> — Page 56, *Applied Predictive Modeling*, 2013.

We can use the `OneHotEncoder` class to implement a dummy encoding as well as a one hot encoding. The `drop` argument can be set to indicate which category will become the one that is assigned all zero values, called the *baseline*. We can set this to *'first'* so that the first category is used. When the labels are sorted alphabetically, the *blue* label will be the first and will become the baseline.

> There will always be one fewer dummy variable than the number of levels. The level with no dummy variable [...] is known as the baseline.
>
> — Page 86, *An Introduction to Statistical Learning with Applications in R*, 2014.

We can demonstrate this with our color categories. The complete example is listed below.

```
# example of a dummy variable encoding
from numpy import asarray
from sklearn.preprocessing import OneHotEncoder
# define data
data = asarray([['red'], ['green'], ['blue']])
print(data)
# define one hot encoding
encoder = OneHotEncoder(drop='first', sparse=False)
# transform data
onehot = encoder.fit_transform(data)
print(onehot)
```

Listing 19.5: Example of demonstrating a dummy variable encoding of color categories.

Running the example first lists the three rows for the categorical variable, then the dummy variable encoding, showing that *green* is encoded as [1, 0], *red* is encoded as [0, 1] and *blue* is encoded as [0, 0] as we specified.

```
[['red']
 ['green']
 ['blue']]
[[0. 1.]
 [1. 0.]
 [0. 0.]]
```

Listing 19.6: Example output from demonstrating a dummy variable encoding of color categories.

Now that we are familiar with the three approaches for encoding categorical variables, let's look at a dataset that has categorical variables.

19.4 Breast Cancer Dataset

We will use the Breast Cancer dataset in this tutorial. This dataset classifies breast cancer patient data as either a recurrence or no recurrence of cancer. There are 286 examples and nine input variables. It is a binary classification problem. You can learn more about this dataset in Chapter 12. We can load this dataset into memory using the Pandas library.

```
...
# load the dataset
dataset = read_csv('breast-cancer.csv', header=None)
# retrieve the array of data
data = dataset.values
```

Listing 19.7: Example of loading the dataset from file.

Once loaded, we can split the columns into input and output for modeling.

```
...
# separate into input and output columns
X = data[:, :-1].astype(str)
y = data[:, -1].astype(str)
```

Listing 19.8: Example of splitting the dataset into input and output elements.

Making use of this function, the complete example of loading and summarizing the raw categorical dataset is listed below.

```
# load and summarize the dataset
from pandas import read_csv
# load the dataset
dataset = read_csv('breast-cancer.csv', header=None)
# retrieve the array of data
data = dataset.values
# separate into input and output columns
X = data[:, :-1].astype(str)
y = data[:, -1].astype(str)
# summarize
print('Input', X.shape)
print('Output', y.shape)
```

Listing 19.9: Example of loading the dataset from file and summarizing the shape.

Running the example reports the size of the input and output elements of the dataset. We can see that we have 286 examples and nine input variables.

```
Input (286, 9)
Output (286,)
```

Listing 19.10: Example output from loading the dataset from file and summarizing the shape.

Now that we are familiar with the dataset, let's look at how we can encode it for modeling.

19.5 OrdinalEncoder Transform

An ordinal encoding involves mapping each unique label to an integer value. This type of encoding is really only appropriate if there is a known relationship between the categories. This relationship does exist for some of the variables in our dataset, and ideally, this should be harnessed when preparing the data. In this case, we will ignore any possible existing ordinal relationship and assume all variables are categorical. It can still be helpful to use an ordinal encoding, at least as a point of reference with other encoding schemes.

We can use the `OrdinalEncoder` from scikit-learn to encode each variable to integers. This is a flexible class and does allow the order of the categories to be specified as arguments if any such order is known. Note that I will leave it as an exercise for you to update the example below to try specifying the order for those variables that have a natural ordering and see if it has an impact on model performance. Once defined, we can call the `fit_transform()` function and pass it to our dataset to create a ordinal transformed version of our dataset.

```
...
# ordinal encode input variables
ordinal = OrdinalEncoder()
```

```
X = ordinal.fit_transform(X)
```

Listing 19.11: Example of transforming a dataset with the OrdinalEncoder.

We can also prepare the target in the same manner.

```
...
# ordinal encode target variable
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
```

Listing 19.12: Example of transforming a dataset with the LabelEncoder.

Let's try it on our breast cancer dataset. The complete example of creating an ordinal encoding transform of the breast cancer dataset and summarizing the result is listed below.

```
# ordinal encode the breast cancer dataset
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
# load the dataset
dataset = read_csv('breast-cancer.csv', header=None)
# retrieve the array of data
data = dataset.values
# separate into input and output columns
X = data[:, :-1].astype(str)
y = data[:, -1].astype(str)
# ordinal encode input variables
ordinal_encoder = OrdinalEncoder()
X = ordinal_encoder.fit_transform(X)
# ordinal encode target variable
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
# summarize the transformed data
print('Input', X.shape)
print(X[:5, :])
print('Output', y.shape)
print(y[:5])
```

Listing 19.13: Example ordinal encoding of the breast cancer dataset.

Running the example transforms the dataset and reports the shape of the resulting dataset. We would expect the number of rows, and in this case, the number of columns, to be unchanged, except all string values are now integer values. As expected, in this case, we can see that the number of variables is unchanged, but all values are now ordinal encoded integers.

```
Input (286, 9)
[[2. 2. 2. 0. 1. 2. 1. 2. 0.]
 [3. 0. 2. 0. 0. 0. 1. 0. 0.]
 [3. 0. 6. 0. 0. 1. 0. 1. 0.]
 [2. 2. 6. 0. 1. 2. 1. 1. 1.]
 [2. 2. 5. 4. 1. 1. 0. 4. 0.]]
Output (286,)
[1 0 1 0 1]
```

Listing 19.14: Example output from ordinal encoding of the breast cancer dataset.

Next, let's evaluate machine learning on this dataset with this encoding. The best practice when encoding variables is to fit the encoding on the training dataset, then apply it to the train and test datasets. We will first split the dataset, then prepare the encoding on the training set, and apply it to the test set.

```
...
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
```

Listing 19.15: Example of splitting the dataset into train and test sets.

We can then fit the `OrdinalEncoder` on the training dataset and use it to transform the train and test datasets.

```
...
# ordinal encode input variables
ordinal_encoder = OrdinalEncoder()
ordinal_encoder.fit(X_train)
X_train = ordinal_encoder.transform(X_train)
X_test = ordinal_encoder.transform(X_test)
```

Listing 19.16: Example of applying the `OrdinalEncoder` to train and test sets without data leakage.

The same approach can be used to prepare the target variable. We can then fit a logistic regression algorithm on the training dataset and evaluate it on the test dataset. The complete example is listed below.

```
# evaluate logistic regression on the breast cancer dataset with an ordinal encoding
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import accuracy_score
# load the dataset
dataset = read_csv('breast-cancer.csv', header=None)
# retrieve the array of data
data = dataset.values
# separate into input and output columns
X = data[:, :-1].astype(str)
y = data[:, -1].astype(str)
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
# ordinal encode input variables
ordinal_encoder = OrdinalEncoder()
ordinal_encoder.fit(X_train)
X_train = ordinal_encoder.transform(X_train)
X_test = ordinal_encoder.transform(X_test)
# ordinal encode target variable
label_encoder = LabelEncoder()
label_encoder.fit(y_train)
y_train = label_encoder.transform(y_train)
y_test = label_encoder.transform(y_test)
# define the model
model = LogisticRegression()
# fit on the training set
```

```
model.fit(X_train, y_train)
# predict on test set
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 19.17: Example of evaluating a model on the breast cancer dataset with an ordinal encoding.

Running the example prepares the dataset in the correct manner, then evaluates a model fit on the transformed data.

Note: Your specific results may vary given the stochastic nature of the learning algorithm, the evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average performance.

In this case, the model achieved a classification accuracy of about 75.79 percent, which is a reasonable score.

```
Accuracy: 75.79
```

Listing 19.18: Example output from evaluating a model on the breast cancer dataset with an ordinal encoding.

Next, let's take a closer look at the one hot encoding.

19.6 OneHotEncoder Transform

A one hot encoding is appropriate for categorical data where no relationship exists between categories. The scikit-learn library provides the `OneHotEncoder` class to automatically one hot encode one or more variables. By default the `OneHotEncoder` will output data with a sparse representation, which is efficient given that most values are 0 in the encoded representation. We will disable this feature by setting the `sparse` argument to False so that we can review the effect of the encoding. Once defined, we can call the `fit_transform()` function and pass it to our dataset to create a quantile transformed version of our dataset.

```
...
# one hot encode input variables
onehot_encoder = OneHotEncoder(sparse=False)
X = onehot_encoder.fit_transform(X)
```

Listing 19.19: Example of transforming a dataset with the `OneHotEncoder`.

As before, we must label encode the target variable. The complete example of creating a one hot encoding transform of the breast cancer dataset and summarizing the result is listed below.

```
# one-hot encode the breast cancer dataset
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
# load the dataset
dataset = read_csv('breast-cancer.csv', header=None)
# retrieve the array of data
```

```
data = dataset.values
# separate into input and output columns
X = data[:, :-1].astype(str)
y = data[:, -1].astype(str)
# one hot encode input variables
onehot_encoder = OneHotEncoder(sparse=False)
X = onehot_encoder.fit_transform(X)
# ordinal encode target variable
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
# summarize the transformed data
print('Input', X.shape)
print(X[:5, :])
```

Listing 19.20: Example one hot encoding of the breast cancer dataset.

Running the example transforms the dataset and reports the shape of the resulting dataset. We would expect the number of rows to remain the same, but the number of columns to dramatically increase. As expected, in this case, we can see that the number of variables has leaped up from 9 to 43 and all values are now binary values 0 or 1.

```
Input (286, 43)
[[0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.
  0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.
  0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0.
  0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0.
  0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  1. 0. 0. 0. 1. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 0.]]
```

Listing 19.21: Example output from one hot encoding of the breast cancer dataset.

Next, let's evaluate machine learning on this dataset with this encoding as we did in the previous section. The encoding is fit on the training set then applied to both train and test sets as before.

```
...
# one-hot encode input variables
onehot_encoder = OneHotEncoder()
onehot_encoder.fit(X_train)
X_train = onehot_encoder.transform(X_train)
X_test = onehot_encoder.transform(X_test)
```

Listing 19.22: Example of applying the `OneHotEncoder` to train and test sets without data leakage.

Tying this together, the complete example is listed below.

```
# evaluate logistic regression on the breast cancer dataset with a one-hot encoding
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
```

```
from sklearn.metrics import accuracy_score
# load the dataset
dataset = read_csv('breast-cancer.csv', header=None)
# retrieve the array of data
data = dataset.values
# separate into input and output columns
X = data[:, :-1].astype(str)
y = data[:, -1].astype(str)
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
# one-hot encode input variables
onehot_encoder = OneHotEncoder()
onehot_encoder.fit(X_train)
X_train = onehot_encoder.transform(X_train)
X_test = onehot_encoder.transform(X_test)
# ordinal encode target variable
label_encoder = LabelEncoder()
label_encoder.fit(y_train)
y_train = label_encoder.transform(y_train)
y_test = label_encoder.transform(y_test)
# define the model
model = LogisticRegression()
# fit on the training set
model.fit(X_train, y_train)
# predict on test set
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 19.23: Example of evaluating a model on the breast cancer dataset with an one hot encoding.

Running the example prepares the dataset in the correct manner, then evaluates a model fit on the transformed data.

Note: Your specific results may vary given the stochastic nature of the learning algorithm, the evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average performance.

In this case, the model achieved a classification accuracy of about 70.53 percent, which is worse than the ordinal encoding in the previous section.

```
Accuracy: 70.53
```

Listing 19.24: Example output from evaluating a model on the breast cancer dataset with an one hot encoding.

19.7 Common Questions

This section lists some common questions and answers when encoding categorical data.