Data Preparation for Machine Learning

Data Cleaning, Feature Selection, and Data Transforms in Python

Jason Brownlee

MACHINE LEARNING MASTERY

Disclaimer

Acknowledgements

Special thanks to my copy editor Sarah Martin and my technical editors Michael Sanderson and Arun Koshy, Andrei Cheremskoy, and John Halfyard.

Copyright

Edition: v1.1

Contents

Preface

Data preparation may be the most important part of a machine learning project. It is the most time consuming part, although it seems to be the least discussed topic. Data preparation, sometimes referred to as data preprocessing, is the act of transforming raw data into a form that is appropriate for modeling. Machine learning algorithms require input data to be numbers, and most algorithm implementations maintain this expectation. As such, if your data contains data types and values that are not numbers, such as labels, you will need to change the data into numbers. Further, specific machine learning algorithms have expectations regarding the data types, scale, probability distribution, and relationships between input variables, and you may need to change the data to meet these expectations.

The philosophy of data preparation is to discover how to best expose the unknown underlying structure of the problem to the learning algorithms. This often requires an iterative path of experimentation through a suite of different data preparation techniques in order to discover what works well or best. The vast majority of the machine learning algorithms you may use on a project are years to decades old. The implementation and application of the algorithms are well understood. So much so that they are routine, with amazing fully featured open-source machine learning libraries like scikit-learn in Python. The thing that is different from project to project is the data. You may be the first person (ever!) to use a specific dataset as the basis for a predictive modeling project. As such, the preparation of the data in order to best present it to the problem of the learning algorithms is the primary task of any modern machine learning project.

The challenge of data preparation is that each dataset is unique and different. Datasets differ in the number of variables (tens, hundreds, thousands, or more), the types of the variables (numeric, nominal, ordinal, boolean), the scale of the variables, the drift in the values over time, and more. As such, this makes discussing data preparation a challenge. Either specific case studies are used, or focus is put on the general methods that can be used across projects. The result is that neither approach is explored. I wrote this book to address the lack of solid advice on data preparation for predictive modeling machine learning projects. I structured the book around the main data preparation activities and designed the tutorials around the most important and widely used data preparation techniques, with a focus on how to use them in the general case so that you can directly copy and paste the code examples into your own projects and get started.

Data preparation is important to machine learning, and I believe that if it is taught at the right level for practitioners, it can be a fascinating, fun, directly applicable, and immeasurably useful toolbox of techniques. I hope that you agree.

Jason Brownlee
2020

Part I

Introduction

Welcome

Welcome to *Data Preparation for Machine Learning*. Data preparation is the process of transforming raw data into a form that is more appropriate for modeling. It may be the most important, most time consuming, and yet least discussed area of a predictive modeling machine learning project. Data preparation is relatively straightforward in principle, although there is a suite of high-level classes of techniques, each with a range of different algorithms, and each appropriate for a specific situation with their own hyperparameters, tips, and tricks. I designed this book to teach you the techniques for data preparation step-by-step with concrete and executable examples in Python.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that may know some applied machine learning. Maybe you know how to work through a predictive modeling problem end-to-end, or at least most of the main steps, with popular tools. The lessons in this book do assume a few things about you, such as:

- You know your way around basic Python for programming.
- You may know some basic NumPy for array manipulation.
- You may know some basic Scikit-Learn for modeling.

This guide was written in the top-down and results-first machine learning style that you're used to from Machine Learning Mastery.

About Your Outcomes

This book will teach you the techniques for data preparation that you need to know as a machine learning practitioner. After reading and working through this book, you will know:

- The importance of data preparation for predictive modeling machine learning projects.
- How to prepare data in a way that avoids data leakage, and in turn, incorrect model evaluation.
- How to identify and handle problems with messy data, such as outliers and missing values.
- How to identify and remove irrelevant and redundant input variables with feature selection methods.

- How to know which feature selection method to choose based on the data types of the variables.
- How to scale the range of input variables using normalization and standardization techniques.
- How to encode categorical variables as numbers and numeric variables as categories.
- How to transform the probability distribution of input variables.
- How to transform a dataset with different variable types and how to transform target variables.
- How to project variables into a lower-dimensional space that captures the salient data relationships.

This book is not a substitute for an undergraduate course in data preparation (if such courses exist) or a textbook for such a course, although it could complement such materials. For a good list of top papers, textbooks, and other resources on data preparation, see the *Further Reading* section at the end of each tutorial.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific method or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them.

About the Book Structure

This book was designed around major data preparation techniques that are directly relevant to real-world problems. There are a lot of things you could learn about data preparation, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials. The tutorials were designed to focus on how to get results with data preparation methods. As such, the tutorials give you the tools to both rapidly understand and apply each technique or operation. There is a mixture of both tutorial lessons and mini-projects to introduce the methods and give plenty of examples and opportunities to practice using them.

Each of the tutorials is designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and

this book is intended to be read and used, not to sit idle. I recommend picking a schedule and sticking to it. The tutorials are divided into six parts; they are:

- **Part 1: Foundation**. Discover the importance of data preparation, tour data preparation techniques, and discover the best practices to use in order to avoid data leakage.
- **Part 2: Data Cleaning**. Discover how to transform messy data into clean data by identifying outliers and identifying and handling missing values with statistical and modeling techniques.
- **Part 3: Feature Selection**. Discover statistical and modeling techniques for feature selection and feature importance and how to choose the technique to use for different variable types.
- **Part 4: Data Transforms**. Discover how to transform variable types and variable probability distributions with a suite of standard data transform algorithms.
- **Part 5: Advanced Transforms**. Discover how to handle some of the trickier aspects of data transforms, such as handling multiple variable types at once, transforming targets, and saving transforms after you choose a final model.
- **Part 6: Dimensionality Reduction**. Discover how to remove input variables by projecting the data into a lower dimensional space with dimensionality-reduction algorithms.

Each part targets a specific learning outcome, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions. The tutorials were not designed to teach you everything there is to know about each of the methods. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- Algorithms were demonstrated on synthetic and small standard datasets to give you the context and confidence to bring the techniques to your own projects.
- Model configurations used were discovered through trial and error and are skillful, but not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties needed beyond the installation of the required packages.

A complete working example is presented with each tutorial for you to inspect and copy-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and is intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Machine learning algorithms are stochastic. This means that they will make different predictions when the same model configuration is trained on the same training data. On top of that, each experimental problem in this book is based on generating stochastic predictions. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the machine learning algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested on a POSIX-compatible machine with Python 3. All code examples will run on modest and modern computer hardware. I am only human, and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it and correct the book (and you can request a free update at any time).

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.
- Books and book chapters.
- Webpages.
- API documentation.
- Open-source projects.

Wherever possible, I have listed and linked to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I have listed those that are first to use a specific technique or first in a specific problem domain. These are not required reading but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on the arXiv pre-print archive. You can search for and download any of the papers listed on Google Scholar Search[1]. Wherever possible, I have tried to link to books on Amazon. I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

[1] https://scholar.google.com

About Getting Help

You might need help along the way. Don't worry; you are not alone.

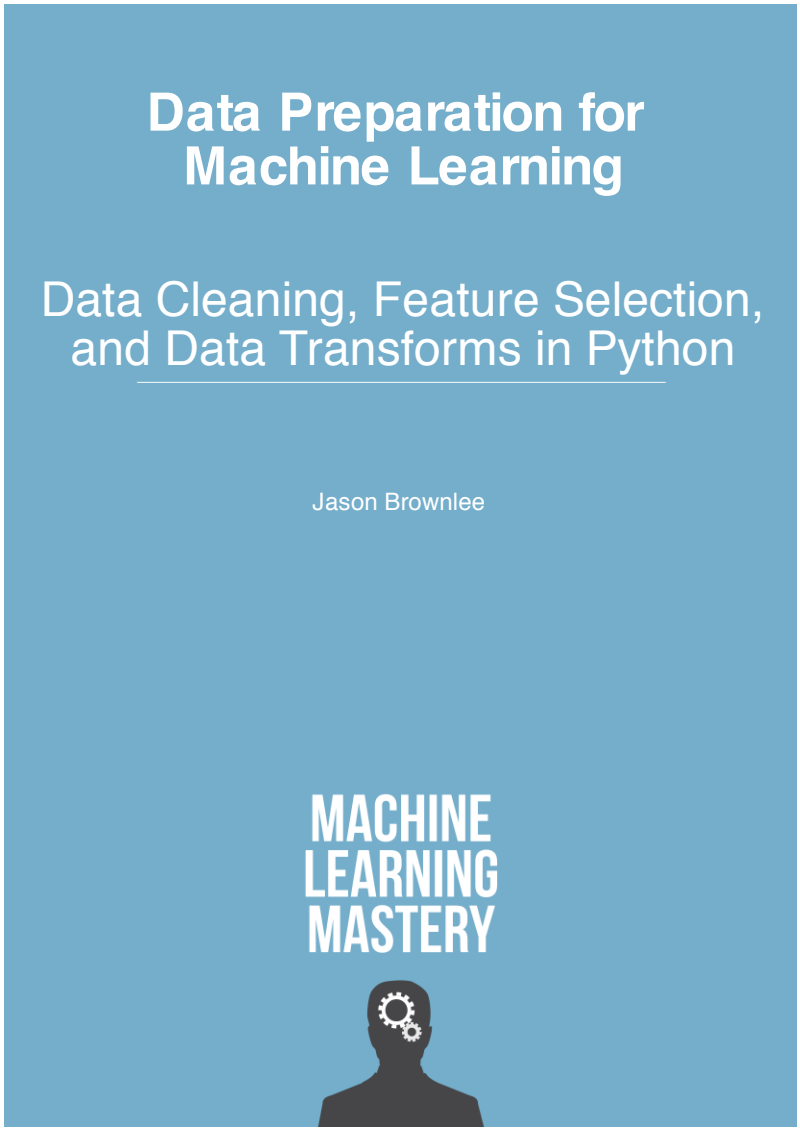
- **Help with a technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.
- **Help with APIs?** If you need help with using a Python library, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help in general?** You can shoot me an email. My details are in *Appendix A*.

Next

Are you ready? Let's dive in!

This is Just a Sample

Thank-you for your interest in **Data Preparation for Machine Learning**.
This is just a sample of the full text. You can purchase the complete book online from:
https://machinelearningmastery.com/data-preparation-for-machine-learning/



Part II

Data Transforms

Chapter 1

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.

1.1 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

1.2 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.

1.3 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.

1.3.1 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:

$$y = \frac{x - \min}{\max - \min} \tag{1.1}$$

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:

$$\begin{aligned} y &= \frac{x - \min}{\max - \min} \\ &= \frac{18.8 - -10}{30 - -10} \\ &= \frac{28.8}{40} \\ &= 0.72 \end{aligned} \tag{1.2}$$

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 1.1: 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 1.2: 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 1.3: Example output from normalizing values in a dataset.

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

1.3.2 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:

$$y = \frac{x - \text{mean}}{\text{standard\_deviation}} \tag{1.3}$$

Where the mean is calculated as:

$$\text{mean} = \frac{1}{N} \times \sum_{i=1}^{N} x_i \tag{1.4}$$

And the standard_deviation is calculated as:

$$\text{standard\_deviation} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \text{mean})^2}{N-1}} \tag{1.5}$$

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:

$$\begin{aligned} y &= \frac{x - \text{mean}}{\text{standard\_deviation}} \\ &= \frac{20.7 - 10}{5} \\ &= \frac{10.7}{5} \\ &= 2.14 \end{aligned} \tag{1.6}$$

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 1.4: 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 1.5: 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.

1.4 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 **??**. 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 1.6: 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 1.7: 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.

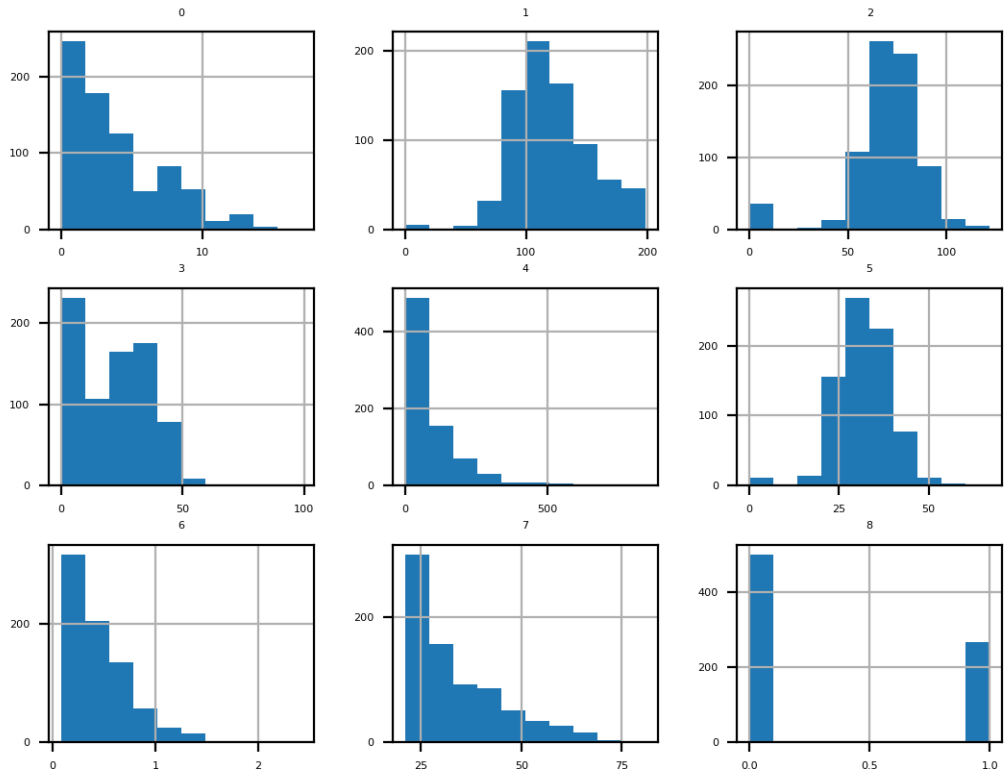


Figure 1.1: Histogram Plots of Input Variables for the Diabetes Binary Classification Dataset.

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 1.8: 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 1.9: Example output from evaluating model performance on the diabetes dataset.

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

1.5 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 1.10: 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 1.11: 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 1.12: 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.

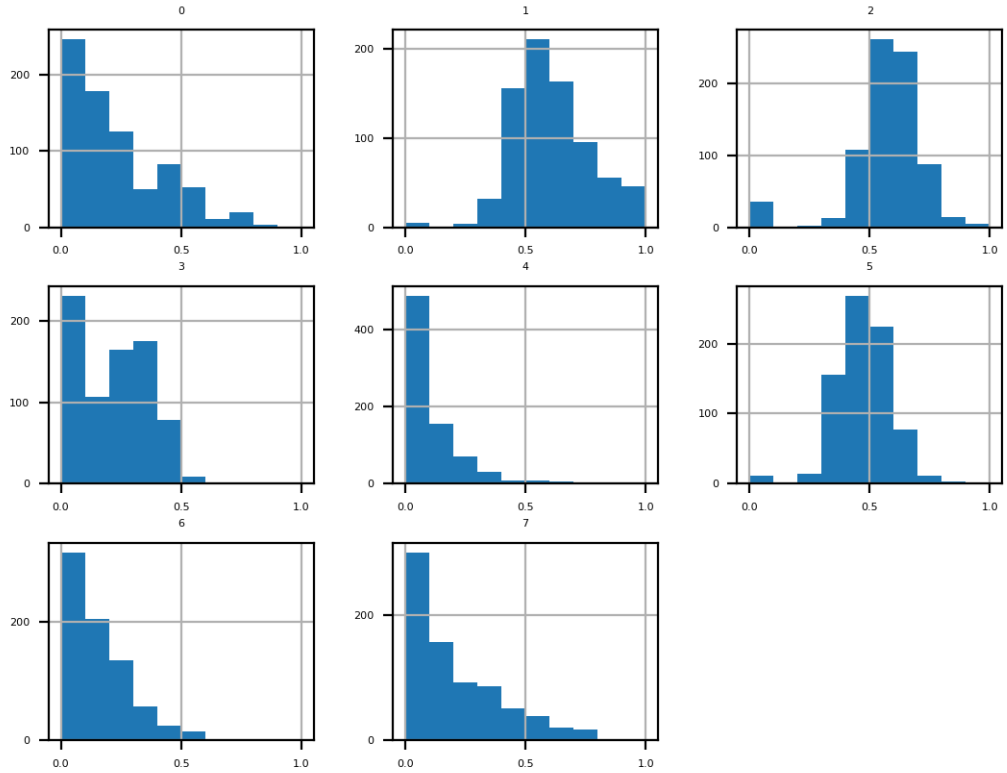


Figure 1.2: Histogram Plots of MinMaxScaler Transformed Input Variables for the Diabetes Dataset.

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 1.13: 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 1.14: Example output from evaluating model performance after a MinMaxScaler transform.

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

1.6 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 1.15: 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 1.16: 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 1.17: 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.

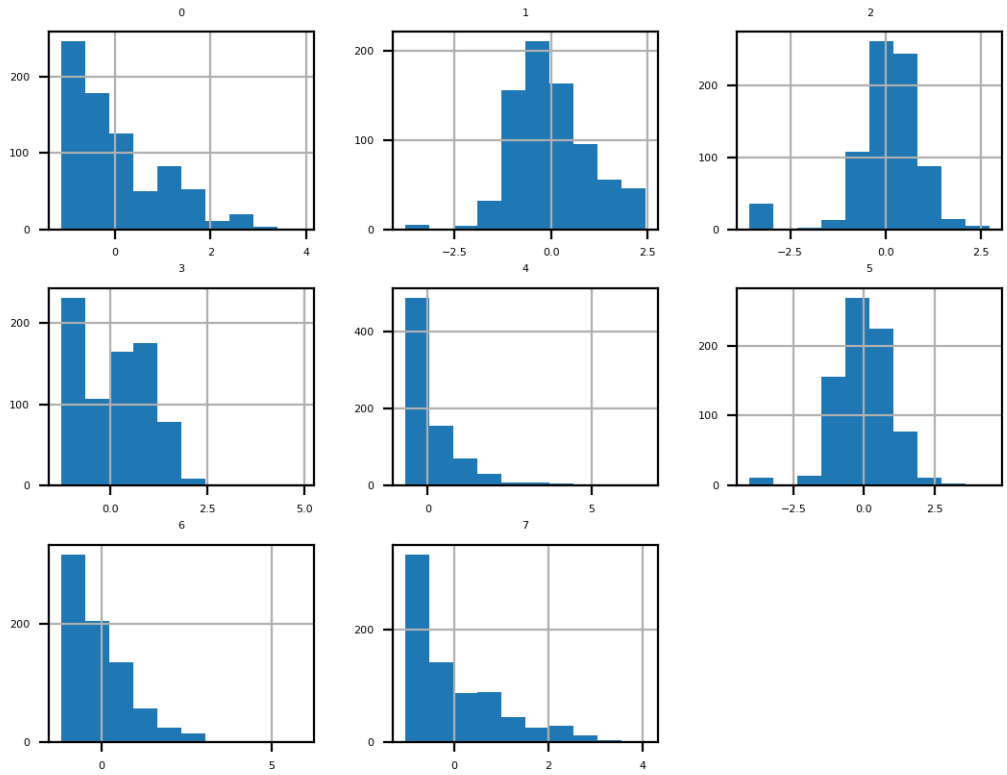


Figure 1.3: Histogram Plots of `StandardScaler` Transformed Input Variables for the Diabetes Dataset.

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 1.18: 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 1.19: Example output from evaluating model performance after a `StandardScaler` transform.

1.7 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.*

Q. Should I Standardize then Normalize?

Standardization can give values that are both positive and negative centered around zero. It may be desirable to normalize data after it has been standardized. This might be a good idea of you have a mixture of standardized and normalized variables and wish all input variables to have the same minimum and maximum values as input for a given algorithm, such as an algorithm that calculates distance measures.

Q. But Which is Best?

This is unknowable. Evaluate models on data prepared with each transform and use the transform or combination of transforms that result in the best performance for your data set on your model.

Q. How Do I Handle Out-of-Bounds Values?

You may normalize your data by calculating the minimum and maximum on the training data. Later, you may have new data with values smaller or larger than the minimum or maximum respectively. One simple approach to handling this may be to check for such out-of-bound values and change their values to the known minimum or maximum prior to scaling. Alternately, you may want to estimate the minimum and maximum values used in the normalization manually based on domain knowledge.

1.8 Further Reading

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

1.8.1 Books

- *Neural Networks for Pattern Recognition*, 1995.
  https://amzn.to/2S8qdwt
- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn
- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/3bbfIAP
- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL

1.8.2 APIs

- `sklearn.preprocessing.MinMaxScaler` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
- `sklearn.preprocessing.StandardScaler` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

1.8.3 Articles

- Should I normalize/standardize/rescale the data? Neural Nets FAQ.
  ftp://ftp.sas.com/pub/neural/FAQ2.html#A_std

1.9 Summary

In this tutorial, you discovered how to use scaler transforms to standardize and normalize numerical input variables for classification and regression. Specifically, you learned:

- 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.

1.9.1 Next

In the next section, we will explore how we can use a robust form of data scaling that is less sensitive to outliers.

This is Just a Sample

Thank-you for your interest in **Data Preparation for Machine Learning**.
This is just a sample of the full text. You can purchase the complete book online from:
https://machinelearningmastery.com/data-preparation-for-machine-learning/