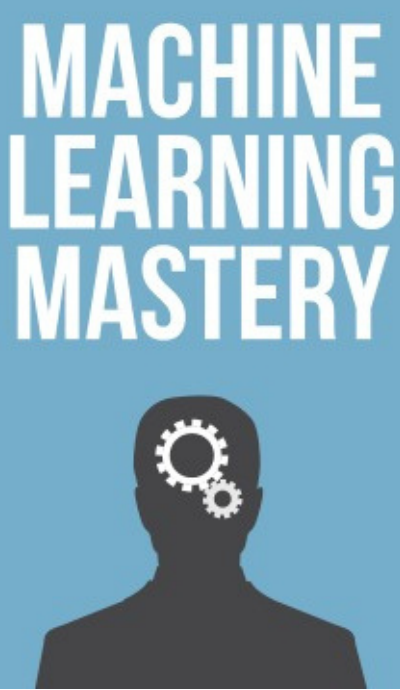
DataPreparationfor

Machine Learning

Data Cleaning, Feature Selection, and Data Transforms in Python



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 pro ject.

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

Intro duction

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 metho ds.

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.

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 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.

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 rep orted.

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]

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 AppliedMachineLearningProcess

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 p erformed.

1.3 WhatIsDataPreparation

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 p erformance.

— 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 HowtoChooseDataPreparationTechniques

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 TutorialOverview

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 WhatIsDatainMachineLearning

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.

prediction.

Input Variables: Columns in the dataset provided to a model in order to make a

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 MachineLearningAlgorithmsExpectNumbers

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 MachineLearningAlgorithmsHaveRequirements

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 ModelPerformanceDependsonData

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.

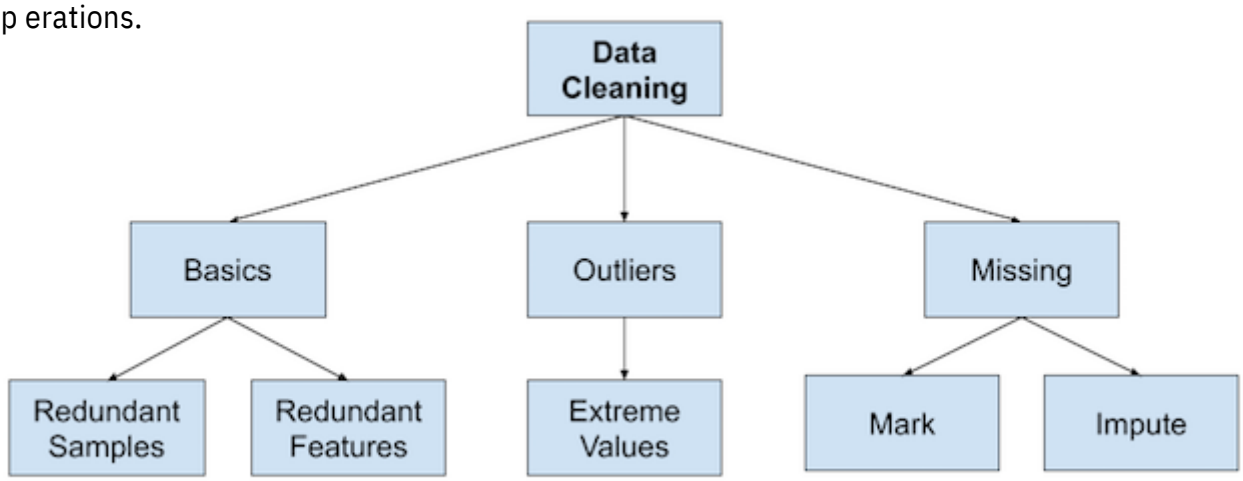


Figure 3.1: Overview of Data Cleaning Techniques.

3.4 FeatureSelection

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).

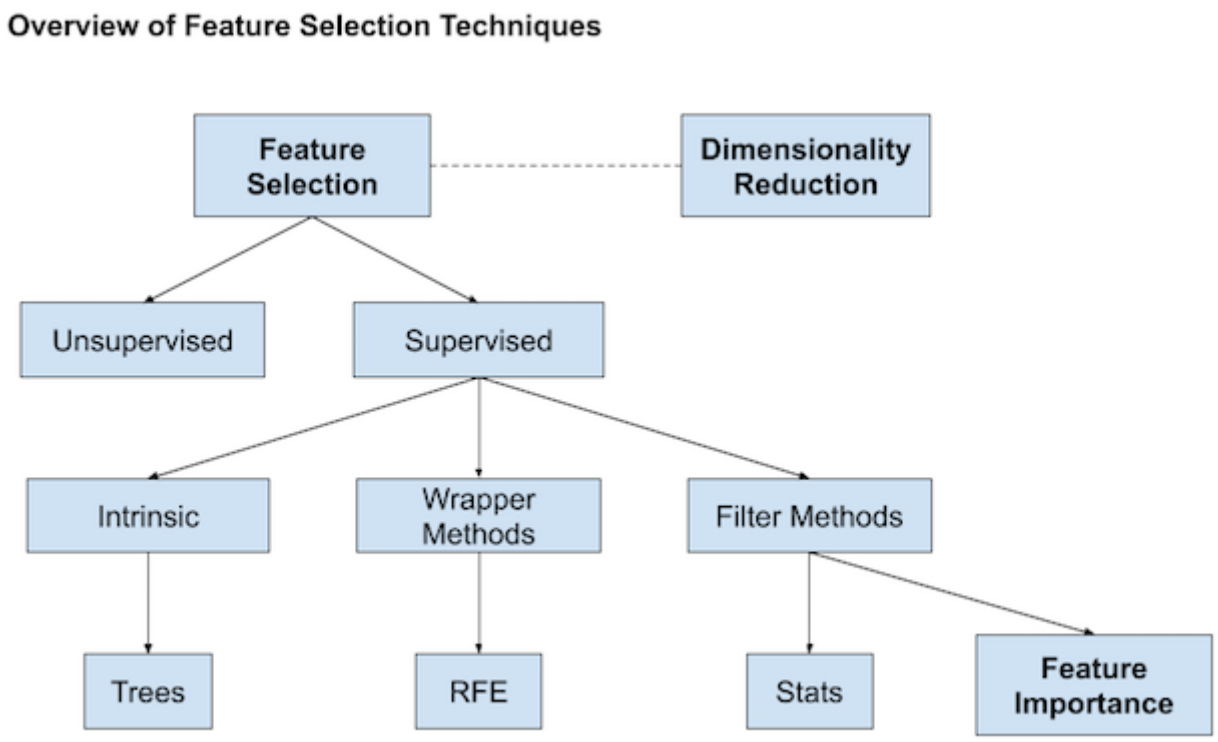


Figure 3.2: Overview of Feature Selection Techniques.

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 DataTransforms

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.

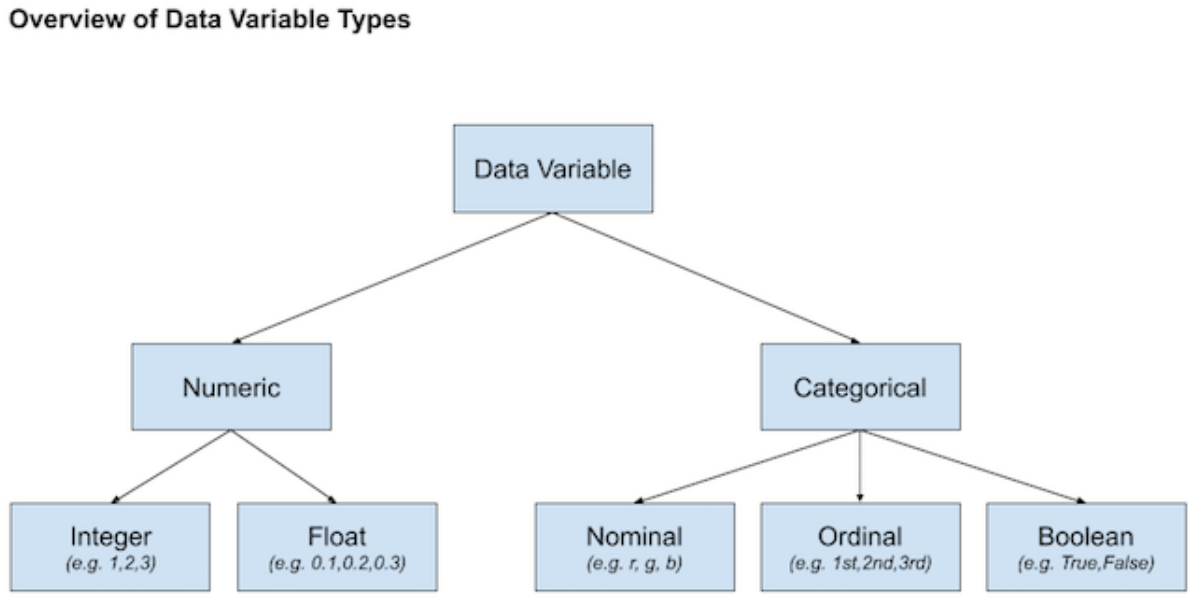


Figure 3.3: Overview of Data Variable Types.

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.
ter 20).

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.

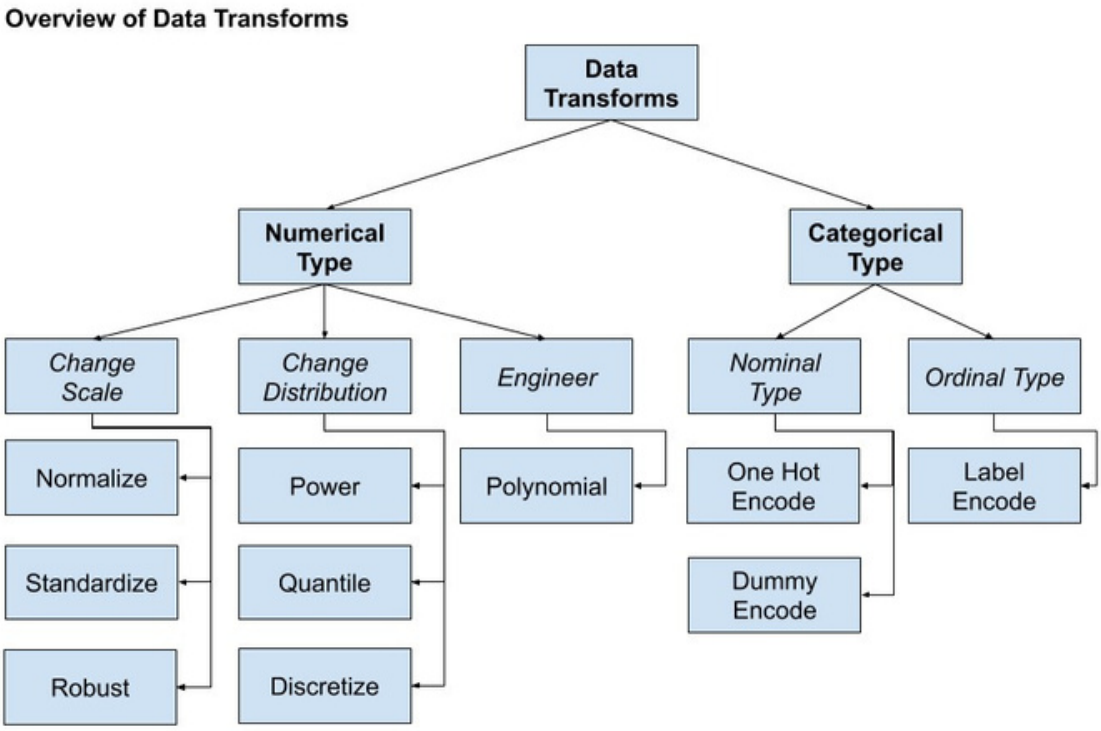


Figure 3.4: Overview of Data Transform Techniques.

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 DimensionalityReduction

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).

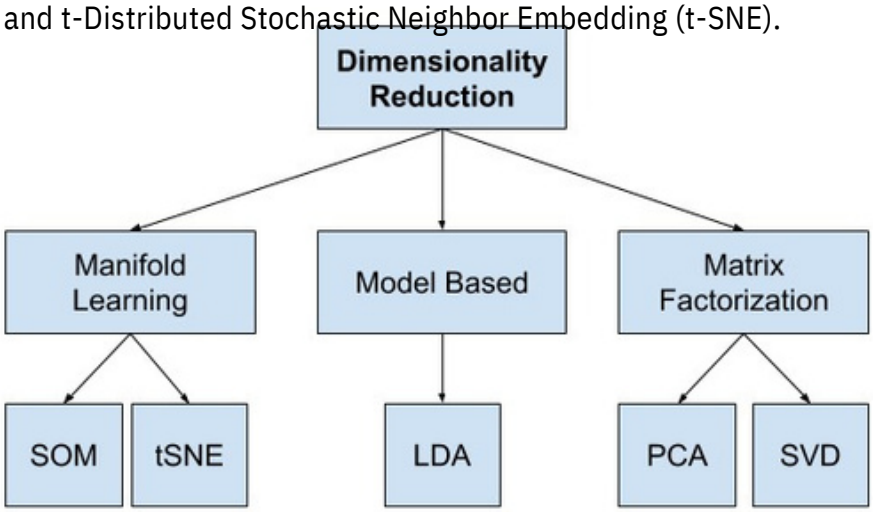


Figure 3.5: Overview of Dimensionality Reduction Techniques.

3.8 Further Reading

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

3.8.1 Bo oks

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 TutorialOverview

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 mo del.

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 p erformance.

> ... 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 DataPreparationWithTrainandTestSets

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 DataPreparationWithk -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 pro cedure.

```
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 Bo oks

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 metho ds.

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 TutorialOverview

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 MessyDatasets

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 OilSpillDataset

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:

ilSpillDataset(oil-spill.csv1 O).

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.

```
0238
1297
2927
3933
4179
5375
6820
7618
8561
957
10 577
1159
1273
13 107
1453
1591
16 893
17 810
18 170
1953
2068
219
221
2392
249
258
269
27 308
28 447
29 392
30 107
3142
324
3345
34 141
```

```
35  110
363
37
758
389
399
40
388
41 220
42
644
43
649
44
499
452
46
937
47 169
48
286
49
492
```

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     23
1     8
2     29
3     7
4     92
5     7
6     93
7     3
8     17
9     9
10    37
11    5
12    82
13    0
14    61
15    8
16    561
17    57
18    57
19    7
20    59
21    73
22    10
      7
      53
      91
      89
      3
      81
      0
      17
      0
```

```
2392
249
258
269
27 308
28 447
29 392
30 107
3142
324
3345
34 141
35 110
363
37 758
389
399
40 388
41 220
42 644
43 649
44 499
452
46 937
47 169
48 286
492
dtype: int64
```

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

5.4 DeleteColumnsThatContainaSingleValue

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 VarianceThresholdto 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.

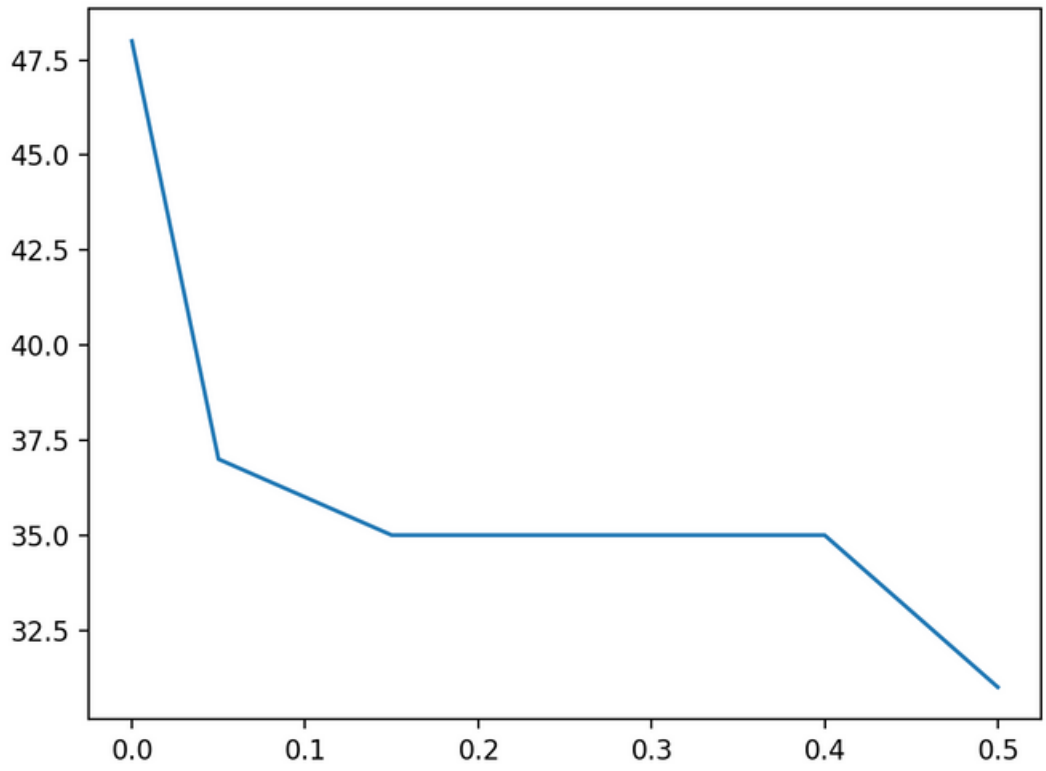


Figure 5.1: Line Plot of Variance Threshold Versus Number of Selected Features.

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 DeleteRowsThatContainDuplicateData

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 Bo oks

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.drōp 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 TutorialOverview

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 WhatareOutliers?

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 xi, which is defined as the number of standard deviations away xi 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 hyp er-rectangle.

6.6 AutomaticOutlierDetection

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:

BostonHousingDataset(1 housing.csv).

BostonHousingDatasetDescription(housin2 g.names).

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 Bo oks

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.htr

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 improve 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 TutorialOverview

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 DiabetesDataset

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 MarkMissingValues

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

```
012...678
count 768.000000 768.000000 768.000000 ... 768.000000 768.000000
768.000000 mean 3.845052120.894531 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%
3.000000117.000000 72.000000... 0.372500 29.000000 0.000000 75%
6.000000140.250000 80.000000... 0.626250 41.000000 1.000000 max
17.000000199.000000122.000000... 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      1  4  8          0 33.6 0.627 50 1
1   7  2      3  5          0 26.6 0.351 31 0
2   1         8  5          0 23.3 0.672 32 1
3   6  6      2  9         94 28.1 0.167 21 0
4   8      1  8  3        168 43.1 2.288 33 1
5   6  4     0             0 25.6 0.201 30 0
6   1  89  66  23  0       88 31.0 0.248 26 1
7   137 40 35 5 116         0 35.3 0.134 29 0
    74 0
    3  78  50  32  10
    115 0 0
```

```
8   2 197 70 45 8      543 30.5 0.158 53 1
9   125 96 0 4 110       0 0.0 0.232 54 1
10  92 0 10 168 74       0 37.6 0.191 30 0
11  0 10 139 80 0 1      0 38.0 0.537 34 1
12  189 60 23 5          0 27.1 1.441 57 0
13  166 72 19          846 30.1 0.398 59 1
14  710000             175 25.8 0.587 51 1
15  0 118 84 47 7        0 30.0 0.484 32 1
16  107 74 0 1 103     230 45.8 0.551 31 1
17  30 38 1 115 70       0 29.6 0.254 31 1
18  30                  83 43.3 0.183 33 0
19                      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.

```
# 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
235
3227
4374
511
6  0
7  0
8  0
d t y p e :  i n t 6 4
```

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    6    1 4 8 . 0    8
0   7 2 . 0      3 5 . 0      N a N    1
1   3 3 . 6    0 . 6 2 7    5 0      1   0
2   8 5 . 0      6 6 . 0      2 9 . 0    1
3   N a N      2 6 . 6      0 . 3 5 1    0
4   3 1      8    1 8 3 . 0    6 4 . 0    1
5   N a N      N a N      2 3 . 3    0
6   0 . 6 7 2    3 2      1    8 9 . 0    1
7   6 6 . 0      2 3 . 0      9 4 . 0    0
    2 8 . 1    0 . 1 6 7    2 1      0
    1 3 7 . 0    4 0 . 0    3 5 . 0
    1 6 8 . 0    4 3 . 1    2 . 2 8 8
    3 3      5    1 1 6 . 0    7 4 . 0
    N a N      N a N      2 5 . 6
    0 . 2 0 1    3 0      3    7 8 . 0
    5 0 . 0      3 2 . 0      8 8 . 0
    3 1 . 0    0 . 2 4 8    2 6      1 0
```

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

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.

```
# 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 Bo oks

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 TutorialOverview

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 HorseColicDataset

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 b elow.

```
# 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...212223  2425  2  2
0  2  .  530101 38.5 66.0 28.0 3.0 ... NaN 2.0 2  11300 0  6  7
1  0  1  534817 39.2 88.0 20.0 NaN ... 2.0 3.0 2  2208 0  0  2
2  1  .  530334 38.3 40.0 24.0 1.0 ... NaN 1.0 2  00  0  2
3  0  1  5290409 39.1 164.0 84.0 4.0 ... 5.3 2.0 1  2208 0  0  1
4  2  .  530255 37.3 104.0 35.0 NaN ... NaN 2.0 2  4300 0  0  1
   0  1                                                    0  2
   1  .
   0  9
   2  .
   0  1
```

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 pro cedure.

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.

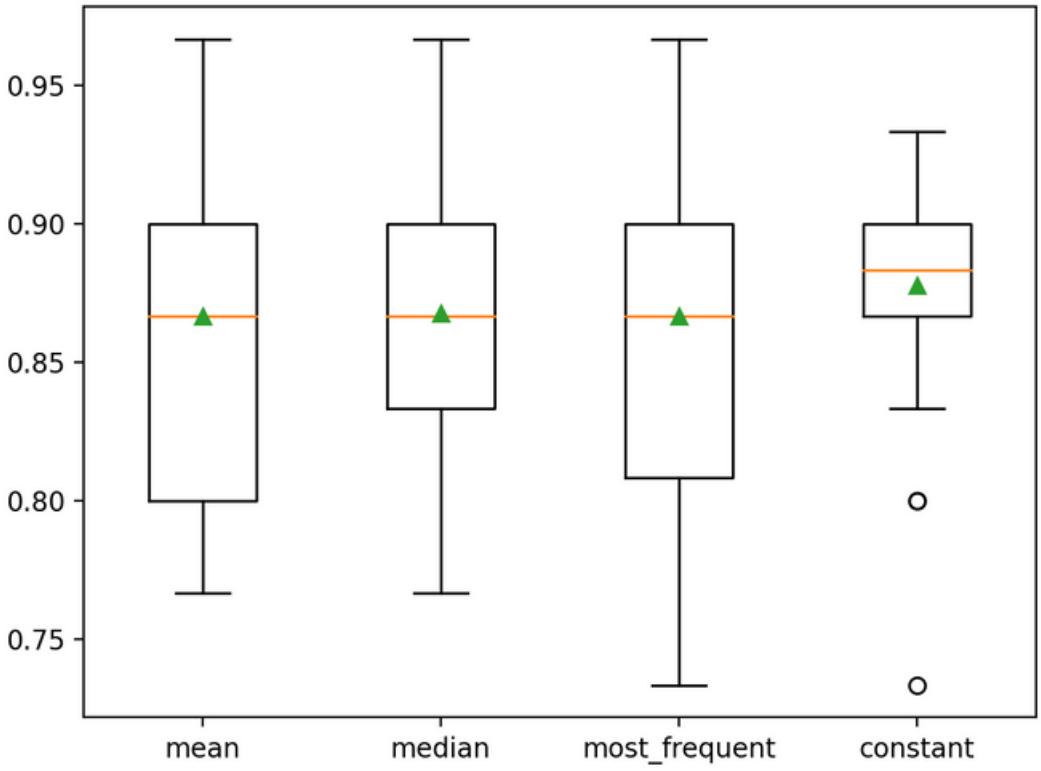


Figure 8.1: Box and Whisker Plot of Statistical Imputation Strategies Applied to the Horse Colic Dataset.

8.4.4 SimpleImputerTransform 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 Bo oks

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 HorseColicDataset

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 b elow.

```
# 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 2223      2425  2   2
0  2  .  530101 38.5  66.0 28.0 3.0 ... NaN 2.0 2  11300 0   6   7
   0  1                                                                   0   2
```

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

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).

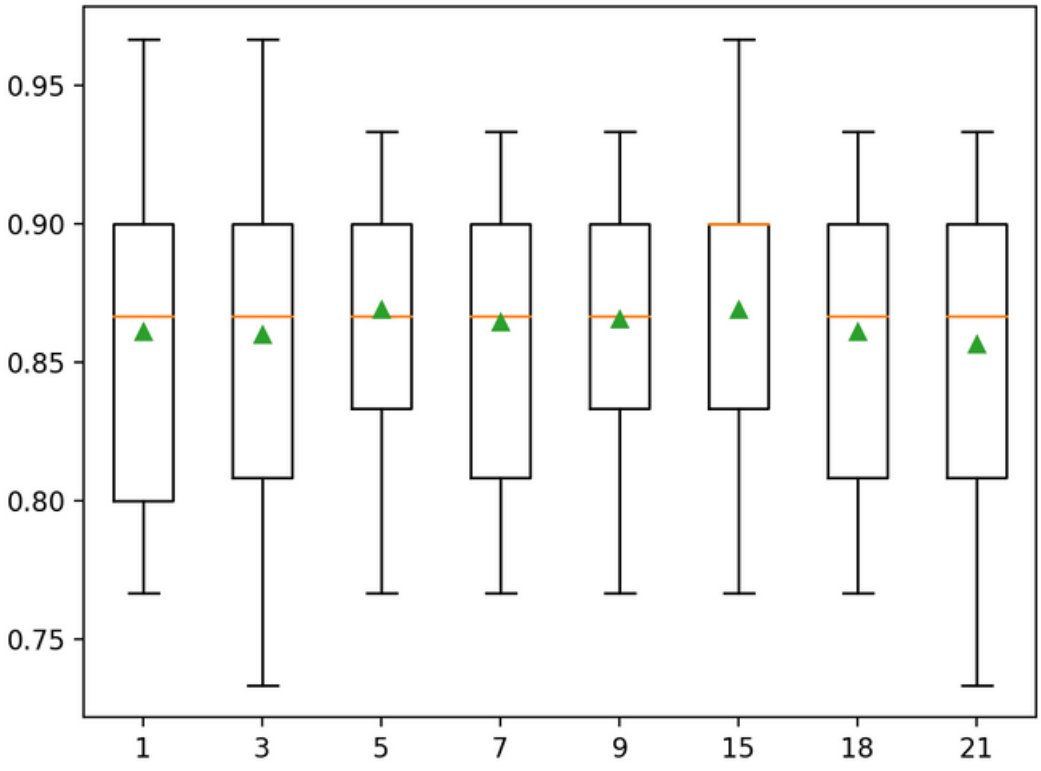


Figure 9.1: Box and Whisker Plot of Imputation Number of Neighbors for the Horse Colic Dataset.

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 Bo oks

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 HorseColicDataset

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 b elow.

```
# 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...212223  2425  2  2
0  2  .  530101 38.5 66.0 28.0 3.0 ... NaN 2.0 2  11300 0  6  7
1  0  1  534817 39.2 88.0 20.0 NaN ... 2.0 3.0 2  2208 0  0  2
2  1  .  530334 38.3 40.0 24.0 1.0 ... NaN 1.0 2  00  0  2
3  0  1  5290409 39.1 164.0 84.0 4.0 ... 5.3 2.0 1  2208 0  0  1
4  2  .  530255 37.3 104.0 35.0 NaN ... NaN 2.0 2  4300 0  0  1
   0  1                                                  0  2
   1  .
   0  9
   2  .
   0  1
```

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 nameIterativeImputer
```

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 pro cedure.

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.

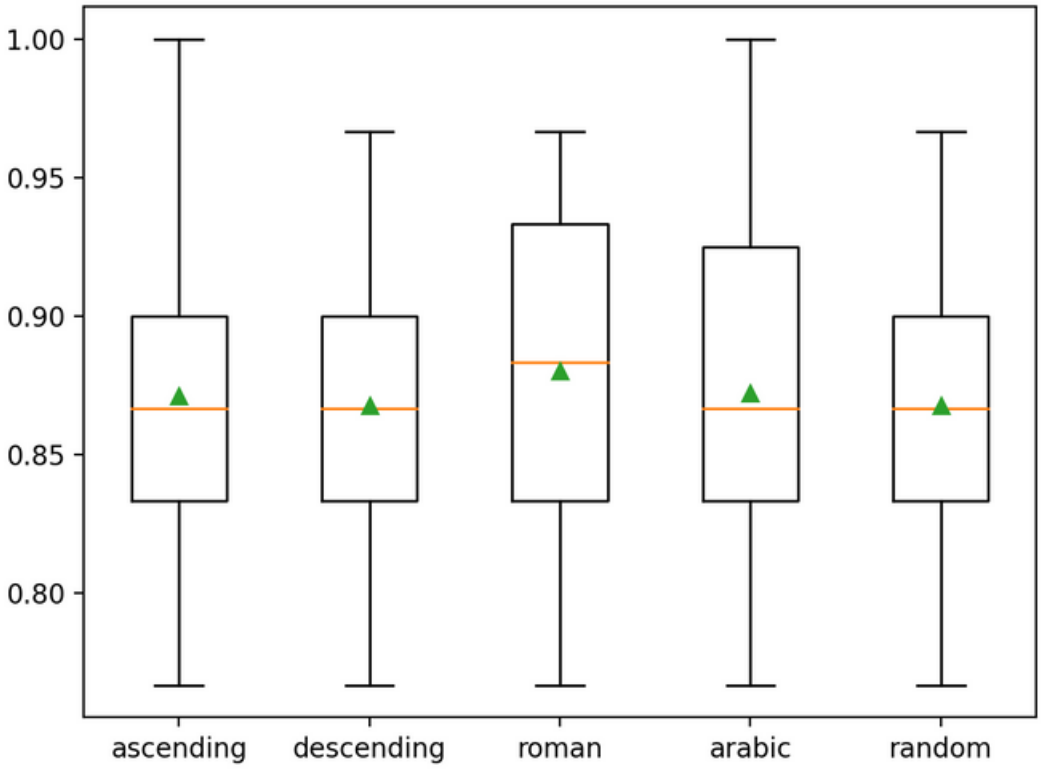


Figure 10.1: Box and Whisker Plot of Imputation Order Strategies Applied to the Horse Colic Dataset.

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.

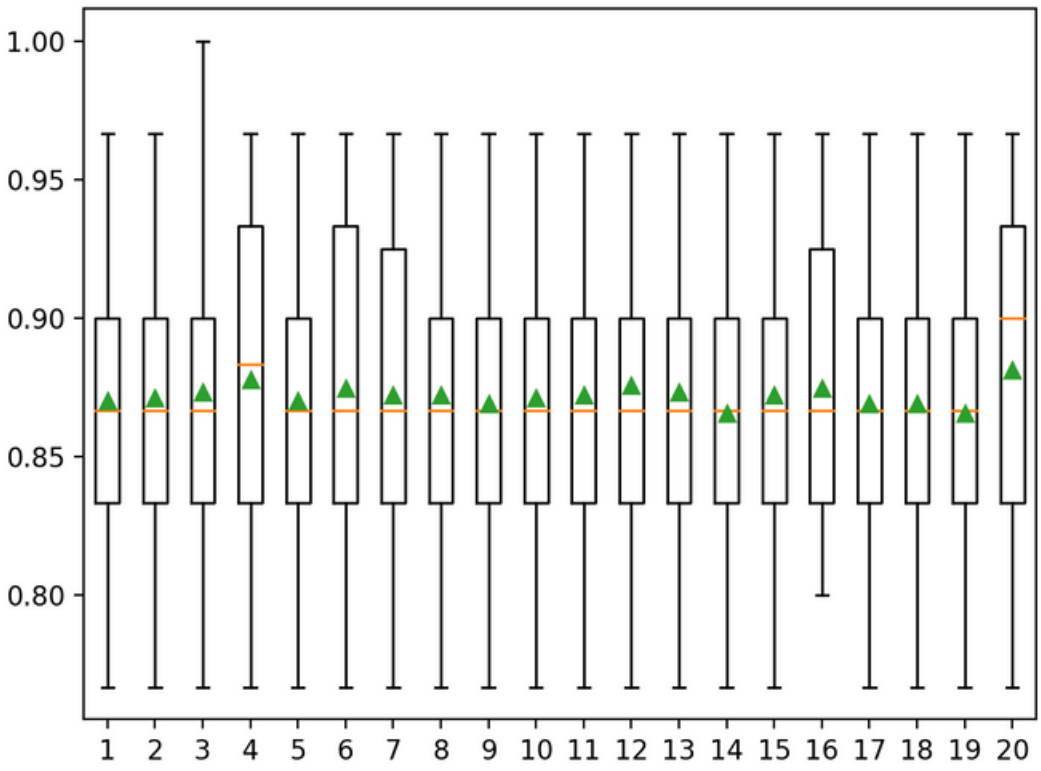


Figure 10.2: Box and Whisker Plot of Number of Imputation Iterations on the Horse Colic Dataset.

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

FeatureSelection

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 TutorialOverview

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 FeatureSelection

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.

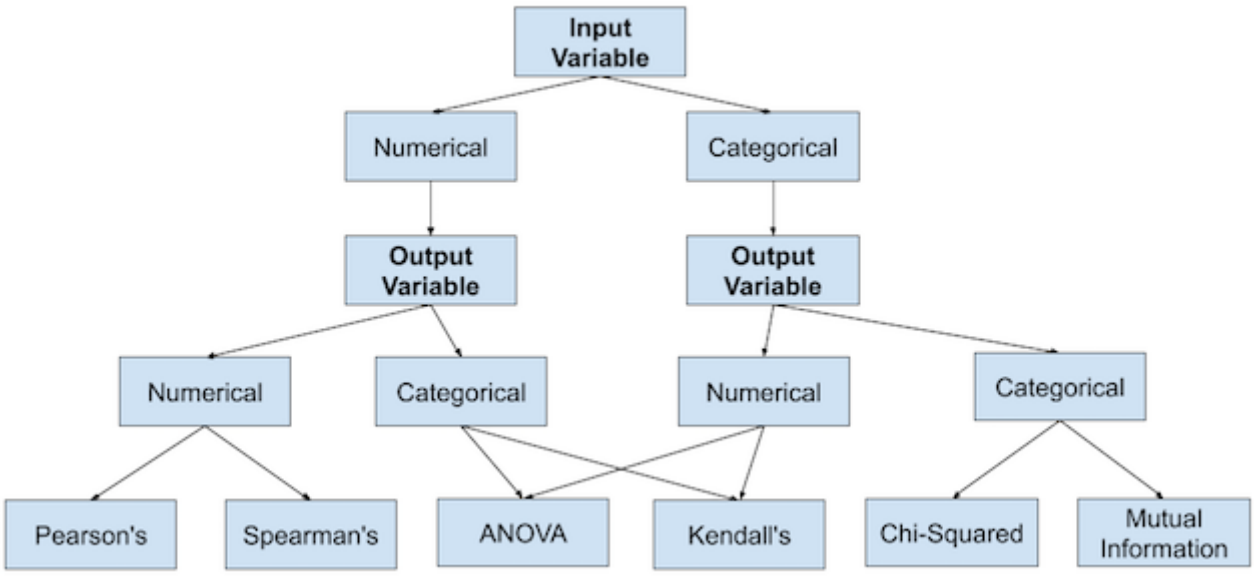


Figure 11.1: How to Choose Feature Selection Methods For Machine Learning.

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

11.3.1 NumericalInput,NumericalOutput

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 Bo oks

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. $\chi 2$, 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 $\chi 2$ (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.

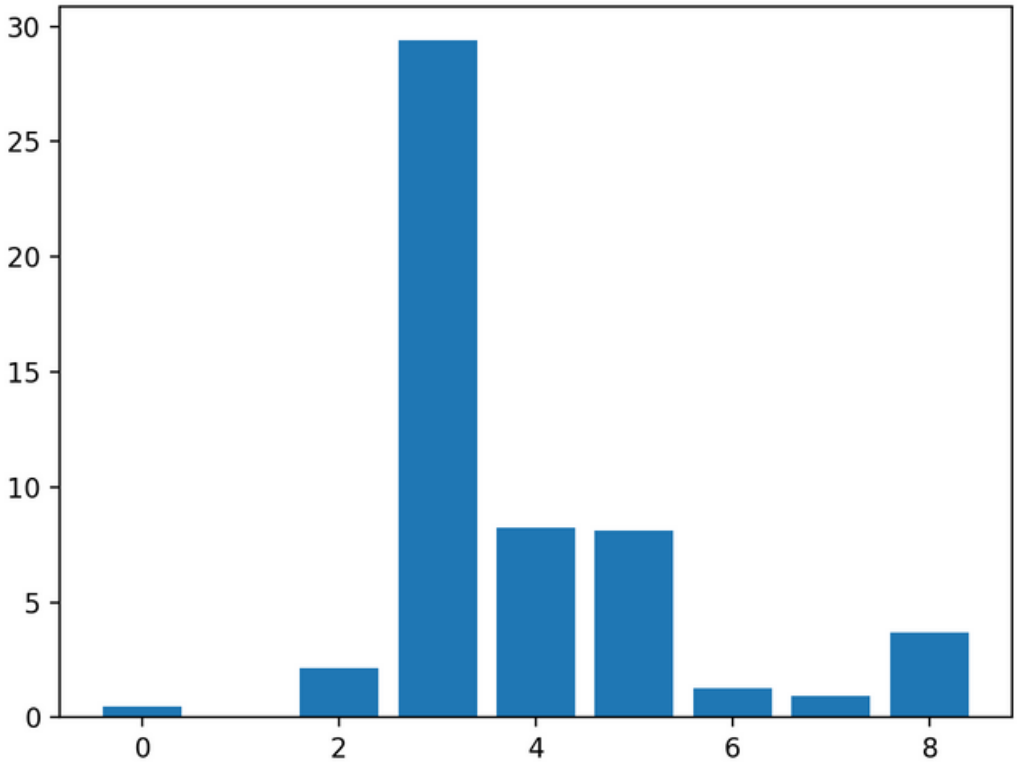


Figure 12.1: Bar Chart of the Input Features vs The Chi-Squared Feature Importance.

12.3.2 MutualInformationFeatureSelection

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.

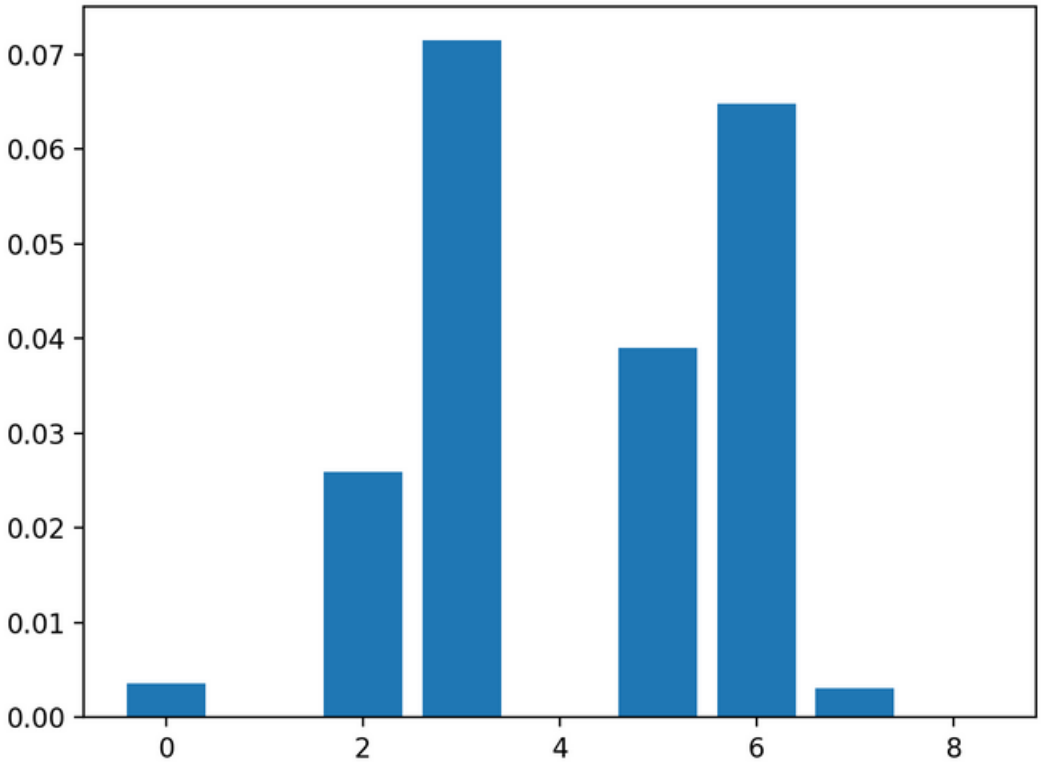


Figure 12.2: Bar Chart of the Input Features vs The Mutual Information Feature Importance.

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 ModelingWithSelectedFeatures

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 ModelBuiltUsingAllFeatures

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 ModelBuiltUsingChi-SquaredFeatures

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 b elow.

```
# 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 Bo oks

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 DiabetesNumericalDataset

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 NumericalFeatureSelection

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

ANOVAF-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.

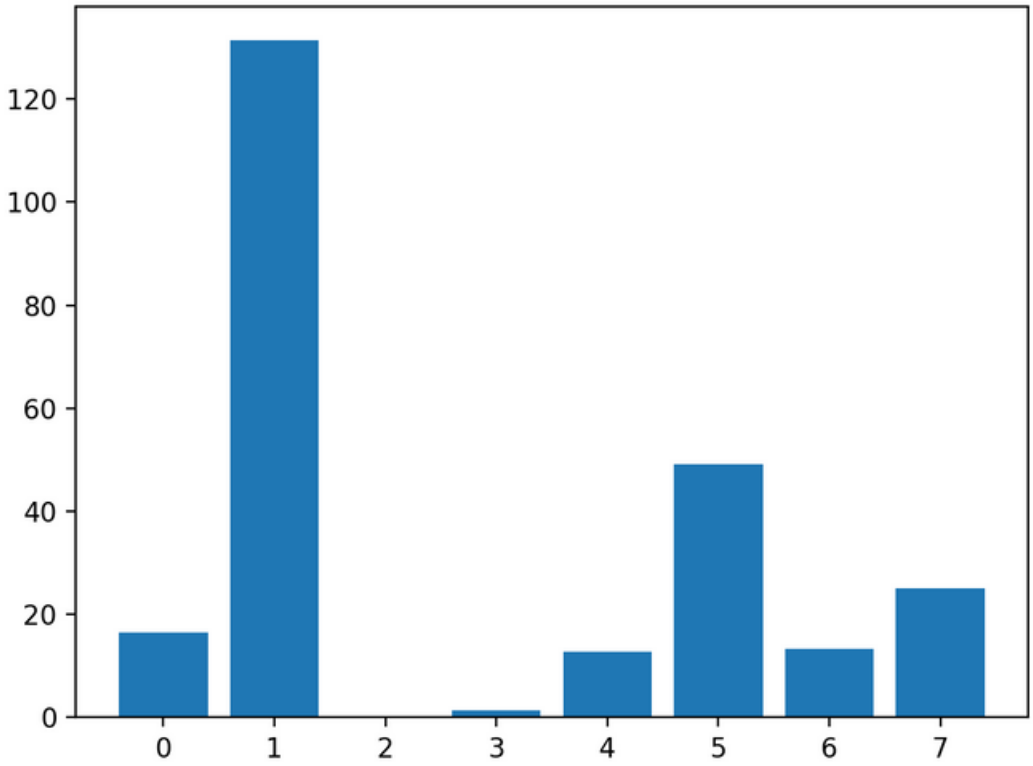


Figure 13.1: Bar Chart of the Input Features vs The ANOVA F-test Feature Importance.

13.3.2 MutualInformationFeatureSelection

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.

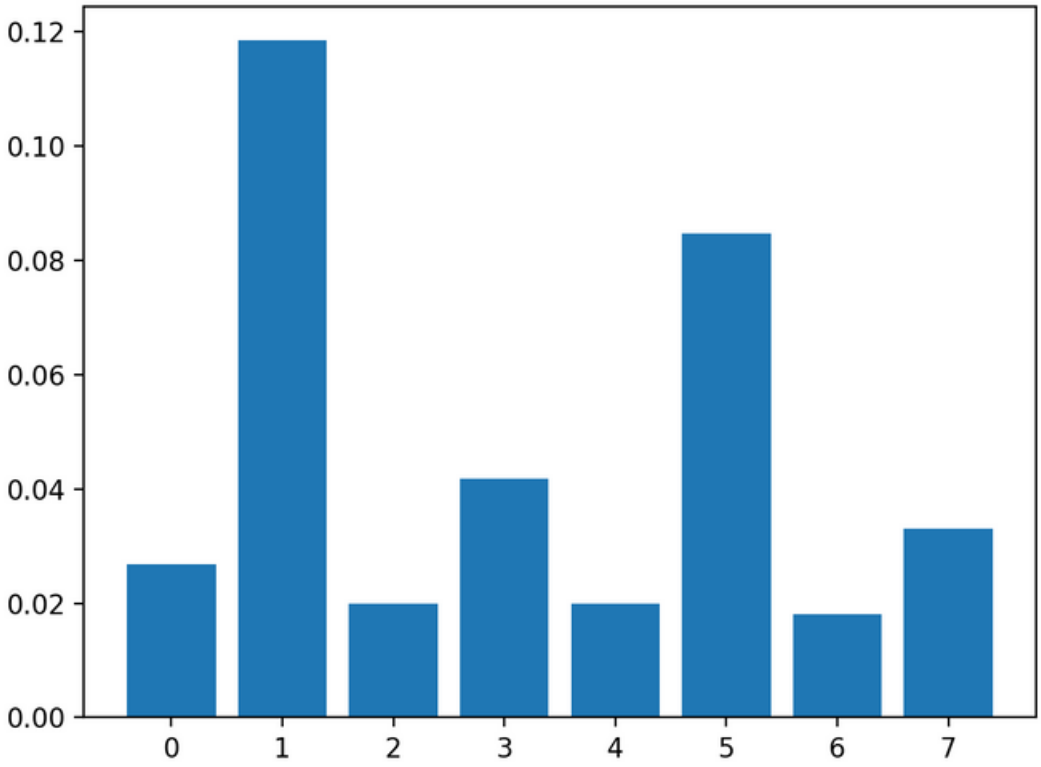


Figure 13.2: Bar Chart of the Input Features vs the Mutual Information Feature Importance.

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 ModelingWithSelectedFeatures

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 ModelBuiltUsingAllFeatures

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 ModelBuiltUsingANOVAF-testFeatures

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 b elow.

```
# 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 namek,separatedbytwounderscores,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 featureswithANOVA.

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.