Ensemble Learning Algorithms With Python

Make Better Predictions with Bagging, Boosting, and Stacking

Jason Brownlee

MACHINE LEARNING MASTERY

Disclaimer

Acknowledgements

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

Copyright

Edition: v1.1

Contents

Preface

Predictive skill can be the most important outcome for some modeling projects. This can be the case when slightly better predictions can result in a large benefit to the organization. A popular example is Netflix, where slightly better recommendations are known to result in better customer retention with the platform. This motivated the one-million-dollar Netflix prize, which was won using a large ensemble of models. On predictive modeling problems where predictive performance is most important, like machine learning competitions, ensembles are almost universally among the top and winning solutions. Ensemble learning algorithms are required if you want the best results.

Ensemble learning used to be an advanced subfield of machine learning, left to the experts. This was for two main reasons. The first is that many ensemble learning algorithms are a more complex type of model, requiring the careful training and integration of multiple other machine learning models. This makes them challenging to implement and challenging to train correctly in a way that avoids data leakage, and in turn, optimistically misleading results. The second reason is that ensemble learning is computationally expensive as instead of fitting and evaluating one model, a single ensemble requires fitting tens, hundreds, or even thousands of models. This used to require large computational resources and expertise with parallel programming.

Thankfully, things have changed. Desktop computers are now incredibly fast and are multi-core by default. We also have access to a suite of advanced ensemble algorithms in modern machine learning libraries such as scikit-Learn in Python, as well as highly efficient third-party implementations of some of the more powerful ensemble algorithms in libraries like XGBoost and LightGBM. It has never been easier to rapidly evaluate advanced ensemble learning algorithms on your own predictive modeling projects. The problem has transformed from a matter of how to implement ensemble methods correctly to instead what are the extent of ensemble methods available and how can they be tailored to specific machine learning projects. That is why I created this book.

I designed this book to take you on a tour of the most effective ensemble machine learning algorithms and show you exactly how they can be used to address classification and regression problems, and how to configure and tune the techniques to get the most out of them. I wanted to skip the theory and math for each method, which may be interesting but do not tell you how to actually configure and use the methods, and focus on showing you exactly how to get a result so that you can bring modern and powerful ensemble learning algorithms to your own projects as fast as possible. Ensemble learning 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 an immeasurably useful toolbox of techniques. I hope that you agree.

Jason Brownlee
2020

Part I

Introduction

Welcome

Welcome to *Ensemble Learning Algorithms With Python*. Ensemble learning algorithms are those techniques that combine the predictions of two or more machine learning algorithms with the goal of improving predictive skill. Ensemble learning algorithms are a more advanced subfield of machine learning, often turned to on machine learning projects when predictive performance is the most important objective. As such, ensembles are widely used by top participants and winners of competitive machine learning competitions.

Traditionally, ensembles have been challenging to implement due to their increased computational cost and complexity, which can introduce data leakage and result in optimistic estimates of model performance. Modern libraries, such as scikit-learn and related third party libraries, now make working with ensembles straightforward for beginners and advanced practitioners alike. I designed this book to teach you the techniques for ensemble learning 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 ensemble learning that you need to know as a machine learning practitioner. After reading and working through this book, you will know:

- The intuition behind drawing upon a crowd or multiple experts when making important decisions and how this intuition carries over to ensemble learning algorithms.

- The benefits of ensemble learning techniques for predictive modeling for both lifting predictive skill and improving model robustness.
- How to develop and evaluate multi-model algorithms for classification and regression problems, providing a precursor to ensemble learning.
- How to develop, configure, and evaluate bagging ensembles for classification and regression predictive modeling problems.
- How to develop and evaluate extensions to bagging, such as random subspace, random forest, and extra trees ensembles.
- How to develop, configure, and evaluate adaptive boosting (AdaBoost) and gradient ensembles for classification and regression predictive modeling problems.
- How to develop and evaluate efficient implementations of gradient boosting ensembles, such as extreme gradient boosting (XGBoost) and light gradient boosting machines (LightGBM).
- How to develop, configure, and evaluate stacking ensembles for classification and regression predictive modeling problems.
- How to develop and evaluate simpler stacking ensembles such as voting and weighted average ensembles.
- How to develop and evaluate extensions to stacking, such as model blending and super learner ensembles.

This book is not a substitute for an undergraduate course on ensemble learning (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 ensemble learning, see the *Further Reading* section at the end of each tutorial.

How to Read This Book

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

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

About the Book Structure

This book was designed around major ensemble learning techniques that are directly relevant to real-world problems. There are a lot of things you could learn about ensemble learning, 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 ensemble learning 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 practical examples to introduce the methods and give plenty of 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 power of ensemble learning techniques, why they are important to getting good performance on your project, and how to develop an intuition for what is being learned.
- **Part 2: Background**: Discover the background required for ensemble learning including the diversity of ensemble members, techniques for combining predictions, the complexity of ensemble models, and the main types of ensemble methods.
- **Part 3: Multiple Models**: Discover machine learning techniques that involve explicitly using multiple models that provide the foundation for ensemble learning methods.
- **Part 4: Bagging**: Discover bootstrap aggregation known as bagging family of ensemble learning techniques including random forest, extra trees, and related methods.
- **Part 5: Boosting**: Discover the boosting family of ensemble learning techniques, including adaptive boosting, gradient boosting, and modern efficient implementations like extreme gradient boosting and light gradient boosting machines.
- **Part 6: Stacking**: Discover the stacked generalization or stacking family of ensemble learning methods, including voting, blending, and related methods.

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

About Python Code Examples

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

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

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

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

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

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

About Further Reading

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

- Research papers.
- Books and book chapters.
- Webpages.
- API documentation.

- Open-source projects.

Wherever possible, I have listed and linked to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I have listed those that are first to use a specific technique or first in a specific problem domain. These are not required reading but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on the arXiv preprint archive. You can search for and download any of the papers listed on Google Scholar Search. 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.

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

What is Ensemble Learning

Many decisions we make in life are based on the opinions of multiple other people. This includes choosing a book to read based on reviews, choosing a course of action based on the advice of multiple medical doctors, and determining guilt. Often, decision making by a group of individuals results in a better outcome than a decision made by any one member of the group. This is generally referred to as the wisdom of the crowd. We can achieve a similar result by combining the predictions of multiple machine learning models for regression and classification predictive modeling problems. This is referred to generally as ensemble machine learning, or simply ensemble learning. In this tutorial, you will discover a gentle introduction to ensemble learning. After reading this tutorial, you will know:

- Many decisions we make involve the opinions or votes of other people.
- The ability of groups of people to make better decisions than individuals is called the wisdom of the crowd.
- Ensemble machine learning involves combining predictions from multiple skillful models.

Let's get started.

1.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Making Important Decisions
2. Wisdom of Crowds
3. Ensemble Machine Learning

1.2 Making Important Decisions

Consider important decisions you make in your life. For example:

- What book to purchase and read next.

- What university to attend.

Candidate books are those that sound interesting, but the book we purchase might have the most favorable reviews. Candidate universities are those that offer the courses we're interested in, but we might choose one based on the feedback from friends and acquaintances that have first-hand experience. We might trust the reviews and star ratings because each individual that contributed a review was (hopefully) unaffiliated with the book and independent of the other people leaving a review. When this is not the case, trust in the outcome is questionable and trust in the system is shaken, which is why Amazon works hard to delete fake reviews for books. Also, consider important decisions we make more personally.

Take, for example the medical treatment for an illness. We take advice from an expert, but we seek a second, third, and even more opinions to confirm we are taking the best course of action. The advice from the second and third opinion may or may not match the first opinion, but we weigh it heavily because it is provided dispassionately, objectively, and independently. If the doctors colluded on their opinion, then we would feel like the process of seeking a second and third opinion has failed.

> ... whenever we are faced with making a decision that has some important consequence, we often seek the opinions of different "experts" to help us make that decision ...
>
> — Page 2, *Ensemble Machine Learning*, 2012.

Finally, consider decisions we make as a society. For example:

- Who should represent a geographical area in a government.
- Whether someone is guilty of a crime.

The democratic election of representatives is based (in some form) on the independent votes of citizens.

> Making decisions based on the input of multiple people or experts has been a common practice in human civilization and serves as the foundation of a democratic society.
>
> — Page v, *Ensemble Methods*, 2012.

An individual's guilt of a serious crime may be determined by a jury of independent peers, often sequestered to enforce the independence of their interpretation. Cases may also be appealed at multiple levels, providing second, third, and more opinions on the outcome.

> The judicial system in many countries, whether based on a jury of peers or a panel of judges, is also based on ensemble-based decision making.
>
> — Pages 1-2, *Ensemble Machine Learning*, 2012.

These are all examples of an outcome arrived at through the combination of lower-level opinions, votes, or decisions.

> ... ensemble-based decision making is nothing new to us; as humans, we use such systems in our daily lives so often that it is perhaps second nature to us.
>
> — Page 1, *Ensemble Machine Learning*, 2012.

In each case, we can see that there are properties of the lower-level decisions that are critical for the outcome to be useful, such as a belief in their independence and that each has some validity on their own. This approach to decision making is so common, it has a name.

1.3 Wisdom of Crowds

This approach to decision making when using humans that make the lower-level decisions is often referred to as the *wisdom of the crowd*. It refers to the case where the opinion calculated from the aggregate of a group of people is often more accurate, useful, or correct than the opinion of any individual in the group. A famous case of this from more than 100 years ago, and often cited, is that of a contest at a fair in Plymouth, England to estimate the weight of an ox. Individuals made their guess and the person whose guess was closest to the actual weight won the meat. The statistician Francis Galton collected all of the guesses afterward and calculated the average of the guesses.

> ... he added all the contestants' estimates, and calculated the mean of the group's guesses. That number represented, you could say, the collective wisdom of the Plymouth crowd. If the crowd were a single person, that was how much it would have guessed the ox weighed.
>
> — Page xiii, *The Wisdom of Crowds*, 2004.

He found that the mean of the guesses made by the contestants was very close to the actual weight. That is, taking the average value of all the numerical weights from the 800 participants was an accurate way of determining the true weight.

> The crowd had guessed that the ox, after it had been slaughtered and dressed, would weigh 1,197 pounds. After it had been slaughtered and dressed, the ox weighed 1,198 pounds. In other words, the crowd's judgment was essentially perfect.
>
> — Page xiii, *The Wisdom of Crowds*, 2004.

This example is given at the beginning of James Surowiecki's 2004 book titled *The Wisdom of Crowds* that explores the ability of groups of humans to make decisions and predictions that are often better than the members of the group.

> This intelligence, or what I'll call "the wisdom of crowds", is at work in the world in many different guises.
>
> — Page xiv, *The Wisdom of Crowds*, 2004.

The book motivates the preference to average the guesses, votes, and opinions of groups of people when making some important decisions instead of searching for and consulting a single expert.

> ... we feel the need to "chase the expert". The argument of this book is that chasing the expert is a mistake [for some problems], and a costly one at that. We should stop hunting and ask the crowd (which, of course, includes the geniuses as well as everyone else) instead. Chances are, it knows.
>
> — Page xv, *The Wisdom of Crowds*, 2004.

The book goes on to highlight a number properties of any system that makes decisions based on groups of people, summarized nicely in Lior Rokach's 2010 book titled *Pattern Classification Using Ensemble Methods* (page 22), as:

- **Diversity of Opinion**: Each member should have private information even if it is just an eccentric interpretation of the known facts.
- **Independence**: Members' opinions are not determined by the opinions of those around them.
- **Decentralization**: Members are able to specialize and draw conclusions based on local knowledge.
- **Aggregation**: Some mechanism exists for turning private judgments into a collective decision.

As a decision-making system, the approach is not always the most effective (e.g. stock market bubbles, fads, etc.), but can be effective in a range of different domains where the outcomes where are important and more precise techniques are unavailable. We can use this approach to decision making in applied machine learning.

1.4 Ensemble Machine Learning

Applied machine learning often involves fitting and evaluating models on a dataset. Given that we cannot know which model will perform best on the dataset beforehand, this may involve a lot of trial and error until we find a model that performs well or best for our project. This is akin to making a decision using a single expert. Perhaps the best expert we can find. A complementary approach is to prepare multiple different models, then combine their predictions. This is called an ensemble machine learning model, or simply an ensemble, and the process of finding a well-performing ensemble model is referred to as *ensemble learning*.

> Ensemble methodology imitates our second nature to seek several opinions before making a crucial decision.
>
> — Page vii, *Pattern Classification Using Ensemble Methods*, 2010.

This is akin to making a decision using the opinions from multiple experts. The most common type of ensemble involves training multiple versions of the same machine learning model in a way that ensures that each ensemble member is different (e.g. decision trees fit on different subsamples of the training dataset), then combining the predictions using averaging or voting. A less common, although just as effective, approach involves training different algorithms on the

same data (e.g. a decision tree, a support vector machine, and a neural network) and combining their predictions.

Like combining the opinions of humans in a crowd, the effectiveness of the ensemble relies on each model having some skill (better than random) and some independence from the other models. This latter point is often interpreted as meaning that the model is skillful in a different way from other models in the ensemble. The hope is that the ensemble results in a better performing model than any contributing member.

> The core principle is to weigh several individual pattern classifiers, and combine them in order to reach a classification that is better than the one obtained by each of them separately.
>
> — Page vii, *Pattern Classification Using Ensemble Methods*, 2010.

At worst, the ensemble limits the worst case of predictions by reducing the variance of the predictions. Model performance can vary with the training data (and the stochastic nature of the learning algorithm in some cases), resulting in better or worse performance for any specific model.

> ... the goal of ensemble systems is to create several classifiers with relatively fixed (or similar) bias and then combining their outputs, say by averaging, to reduce the variance.
>
> — Page 2, *Ensemble Machine Learning*, 2012.

An ensemble can smooth this out and ensure that predictions made are closer to the average performance of contributing members. Further, reducing the variance in predictions often results in a lift in the skill of the ensemble. This comes at the added computational cost of fitting and maintaining multiple models instead of a single model. Although ensemble predictions will have a lower variance, they are not guaranteed to have better performance than any single contributing member.

> ... researchers in the computational intelligence and machine learning community have studied schemes that share such a joint decision procedure. These schemes are generally referred to as ensemble learning, which is known to reduce the classifiers' variance and improve the decision system's robustness and accuracy.
>
> — Page v, *Ensemble Methods*, 2012.

Sometimes, the best performing model, e.g. the best expert, is sufficiently superior compared to other models that combining its predictions with other models can result in worse performance. As such, selecting models, even ensemble models, still requires carefully controlled experiments with a robust test harness.

1.5 Further Reading

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

Books

- *The Wisdom of Crowds*, 2004.
  https://amzn.to/2UNOMll
- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Ensemble learning, Scholarpedia.
  http://www.scholarpedia.org/article/Ensemble_learning
- Wisdom of the crowd, Wikipedia.
  https://en.wikipedia.org/wiki/Wisdom_of_the_crowd
- The Wisdom of Crowds, Wikipedia.
  https://en.wikipedia.org/wiki/The_Wisdom_of_Crowds

1.6 Summary

In this tutorial, you discovered a gentle introduction to ensemble learning. Specifically, you learned:

- Many decisions we make involve the opinions or votes of other people.
- The ability of groups of people to make better decisions than individuals is called the wisdom of the crowd.
- Ensemble machine learning involves combining predictions from multiple skillful models.

Next

In the next section, we will review the importance of ensemble algorithms in machine learning and why they are required.

Chapter 2

Why Use Ensemble Learning

Ensembles are predictive models that combine predictions from two or more models. Ensemble learning methods are popular and the go-to technique when the best performance on a predictive modeling project is the most important outcome. Nevertheless, they are not always the most appropriate technique to use and beginners the field of applied machine learning have the expectation that ensembles or a specific ensemble method are always the best method to use. Ensembles offer two specific benefits on a predictive modeling project, and it is important to know what these benefits are and how to measure them to ensure that using an ensemble is the right decision on your project. In this tutorial, you will discover the benefits of using ensemble methods for machine learning. After reading this tutorial, you will know:

- A minimum benefit of using ensembles is to reduce the spread in the average skill of a predictive model.
- A key benefit of using ensembles is to improve the average prediction performance over any contributing member in the ensemble.
- The mechanism for improved performance with ensembles is often the reduction in the variance component of prediction errors made by the contributing models.

Let's get started.

2.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Ensemble Learning
2. Use Ensembles to Improve Robustness
3. Bias, Variance, and Ensembles
4. Use Ensembles to Improve Performance

2.2 Benefits of Ensemble Learning

An ensemble is a machine learning model that combines the predictions from two or more models. The models that contribute to the ensemble, referred to as ensemble members, may be the same type or different types and may or may not be trained on the same training data. The predictions made by the ensemble members may be combined using statistics, such as the mode or mean, or by more sophisticated methods that learn how much to trust each member and under what conditions.

The study of ensemble methods really picked up in the 1990s, and that decade was when papers on the most popular and widely used methods were published, such as core bagging, boosting, and stacking methods. In the late 2000s, adoption of ensembles picked up due in part to their huge success in machine learning competitions, such as the Netflix prize and later competitions on Kaggle.

> Over the last couple of decades, multiple classifier systems, also called ensemble systems have enjoyed growing attention within the computational intelligence and machine learning community.
>
> — Page 1, *Ensemble Machine Learning*, 2012.

Ensemble methods greatly increase computational cost and complexity. This increase comes from the expertise and time required to train and maintain multiple models rather than a single model. This forces the question: **Why should we consider using an ensemble?** There are two main reasons to use an ensemble over a single model, and they are related; they are:

- **Performance** An ensemble can make better predictions and achieve better performance than any single contributing model.
- **Robustness**: An ensemble reduces the spread or dispersion of the predictions and model performance.

Ensembles are used to achieve better predictive performance on a predictive modeling problem than a single predictive model. The way this is achieved can be understood as the model reducing the variance component of the prediction error by adding bias (i.e. in the context of the bias-variance trade-off).

> Originally developed to reduce the variance — thereby improving the accuracy — of an automated decision-making system ...
>
> — Page 1, *Ensemble Machine Learning*, 2012.

There is another important and less discussed benefit of ensemble methods, which is the improved robustness or reliability in the average performance of the model. These are both important concerns on a machine learning project and sometimes we may prefer one or both properties from a model. Let's take a closer look at these two properties in order to better understand the benefits of using ensemble learning on a project.

2.3 Use Ensembles to Improve Robustness

On a predictive modeling project, we often evaluate multiple models or modeling pipelines and choose one that performs well or best as our final model. The algorithm or pipeline is then fit on all available data and used to make predictions on new data. We have an idea of how well the model will perform on average from our test harness, typically estimated using repeated k-fold cross-validation as a gold standard. The problem is, average performance might not be sufficient. An average accuracy or error of a model is a summary of the expected performance, when in fact, some models performed better and some models performed worse on different subsets of the data.

The standard deviation is the average difference between an observation and the mean and summarizes the dispersion or spread of data. For an accuracy or error measure of a model's performance, it can give you an idea of the spread of the model's behavior or capability. Looking at the minimum and maximum model performance scores will give you an idea of the worst and best performance you might expect from the model, and this might not be acceptable for your application.

The simplest ensemble is to fit the model multiple times on the training datasets and combine the predictions using a summary statistic, such as the mean for regression or the mode for classification. Importantly, each model needs to be slightly different. These differences may be caused by the stochastic learning algorithm, difference in the composition of the training dataset, or differences in the model itself. This will reduce the spread in the predictions made by the model. The mean performance will probably be about the same, although the worst-case and best-case performance will be brought closer to the mean performance. In effect, it smooths out the expected performance of the model. We can refer to this as the *robustness* in the expected performance of the model and is a minimum benefit of using an ensemble method. An ensemble may or may not improve modeling performance over any single contributing member, but at minimum, it should reduce the spread in the average performance of the model.

2.4 Bias, Variance, and Ensembles

Machine learning models for classification and regression learn a mapping function from inputs to outputs. This mapping is learned from examples from the problem domain, the training dataset, and is evaluated on data not used during training, the test dataset. The errors made by a machine learning model are often described in terms of two properties: the bias and the variance.

The bias is a measure of how close the model can capture the mapping function between inputs and outputs. It captures the rigidity of the model: the strength of the assumption the model has about the functional form of the mapping between inputs and outputs. The variance of the model is the amount the performance of the model changes when it is fit on different training data. It captures the impact of the specifics of the data has on the model.

> Variance refers to the amount by which [the model] would change if we estimated it using a different training data set.
>
> — Page 34, *An Introduction to Statistical Learning with Applications in R*, 2014.

The bias and the variance of a model's performance are connected. Ideally, we would prefer a model with low bias and low variance, although in practice, this is very challenging. In fact, this could be described as the goal of applied machine learning for a given predictive modeling problem. Reducing the bias can often easily be achieved by increasing the variance. Conversely, reducing the variance can easily be achieved by increasing the bias.

> This is referred to as a trade-off because it is easy to obtain a method with extremely low bias but high variance [...] or a method with very low variance but high bias ...
>
> — Page 36, *An Introduction to Statistical Learning with Applications in R*, 2014.

Some models naturally have a high bias or a high variance, which can be often relaxed or increased using hyperparameters that change the learning behavior of the algorithm. Ensembles provide a way to reduce the variance of the predictions; that is the amount of error in the predictions made that can be attributed to *variance*. This is not always the case, but when it is, this reduction in variance, in turn, leads to improved predictive performance.

> Empirical and theoretical evidence show that some ensemble techniques (such as bagging) act as a variance reduction mechanism, i.e., they reduce the variance component of the error. Moreover, empirical results suggest that other ensemble techniques (such as AdaBoost) reduce both the bias and the variance parts of the error.
>
> — Page 39, *Pattern Classification Using Ensemble Methods*, 2010.

Using ensembles to reduce the variance properties of prediction errors leads to the key benefit of using ensembles in the first place: to improve predictive performance.

2.5 Use Ensembles to Improve Performance

Reducing the variance element of the prediction error improves predictive performance. We explicitly use ensemble learning to seek better predictive performance, such as lower error on regression or high accuracy for classification.

> ... there is a way to improve model accuracy that is easier and more powerful than judicious algorithm selection: one can gather models into ensembles.
>
> — Page 2, *Ensemble Methods in Data Mining*, 2010.

This is the primary use of ensemble learning methods and the benefit demonstrated through the use of ensembles by the majority of winners of machine learning competitions, such as the Netflix prize and competitions on Kaggle.

> In the Netflix Prize, a contest ran for two years in which the first team to submit a model improving on Netflix's internal recommendation system by 10% would win $1,000,000. [...] the final edge was obtained by weighing contributions from the models of up to 30 competitors.

— Page 8, *Ensemble Methods in Data Mining*, 2010.

This benefit has also been demonstrated with academic competitions, such as top solutions for the famous ImageNet dataset in computer vision.

> An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task.

— *Deep Residual Learning for Image Recognition*, 2015.

When used in this way, an ensemble should only be adopted if it performs better on average than any contributing member of the ensemble. If this is not the case, then the contributing member that performs better should be used instead. Consider the distribution of expected scores calculated by a model on a test harness, such as repeated k-fold cross-validation, as we did above when considering the *robustness* offered by an ensemble. An ensemble that reduces the variance in the error, in effect, will shift the distribution of estimated performance scores rather than simply shrink the spread of the distribution. This can result in a better average performance as compared to any single model.

This is not always the case, and having this expectation is a common mistake made by beginners. It is possible, and even common, for the performance of an ensemble to perform no better than the best-performing member of the ensemble. This can happen if the ensemble has one top-performing model and the other members do not offer any benefit or the ensemble is not able to harness their contribution effectively. It is also possible for an ensemble to perform worse than the best-performing member of the ensemble. This too is common and typically involves one top-performing model whose predictions are made worse by one or more poor-performing other models and the ensemble is not able to harness their contributions effectively. As such, it is important to test a suite of ensemble methods and tune their behavior, just as we do for any individual machine learning model.

2.6 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5
- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/3frGM1A

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Ensemble learning, Scholarpedia.
  http://www.scholarpedia.org/article/Ensemble_learning

2.7 Summary

In this tutorial, you discovered the benefits of using ensemble methods for machine learning. Specifically, you learned:

- A minimum benefit of using ensembles is to reduce the spread in the average skill of a predictive model.
- A key benefit of using ensembles is to improve the average prediction performance over any contributing member in the ensemble.
- The mechanism for improved performance with ensembles is often the reduction in the variance component of prediction errors made by the contributing models.

Next

In the next section, we will develop an intuition for what ensemble learning algorithms are doing for classification and regression problems.

Chapter 3

Intuition for Ensemble Learning

Ensembles are a machine learning method that combine the predictions from multiple models in an effort to achieve better predictive performance. There are many different types of ensembles, although all approaches have two key properties: they require that the contributing models are different so that they make different errors and they combine the predictions in an attempt to harness what each different model does well. Nevertheless, it is not clear how ensembles manage to achieve this, especially in the context of classification and regression type predictive modeling problems. It is important to develop an intuition for what exactly ensembles are doing when they combine predictions as it will help choose and configure appropriate models on predictive modeling projects. In this tutorial, you will discover the intuition behind how ensemble learning methods work. After reading this tutorial, you will know:

- Ensemble learning methods work by in-effect combining the mapping functions learned by contributing members.
- Ensembles for classification are best understood by the combination of decision boundaries of members.
- Ensembles for regression are best understood by the combination of hyperplanes of members.

Let's get started.

3.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. How Do Ensembles Work
2. Intuition for Classification Ensembles
3. Intuition for Regression Ensembles

3.2 How Do Ensembles Work

Ensemble learning refers to combining the predictions from two or more models. The goal of using ensemble methods is to improve the skill of predictions over that of any of the contributing members. This objective is straightforward but it is less clear how exactly ensemble methods are able to achieve this. It is important to develop an intuition for how ensemble techniques work as it will help you both choose and configure specific ensemble methods for a prediction task and interpret their results to come up with alternative ways to further improve performance. Consider a simple ensemble that trains two models on slightly different samples of the training dataset and averages their predictions.

Each of the member models can be used in a standalone manner to make predictions, although the hope is that averaging their predictions improves their performance. This can only be the case if each model makes different predictions. Different predictions mean that in some specific cases, model 1 will perform better and model 2 will perform worse, and the reverse for other specific cases. Averaging their predictions seeks to reduce these errors across the predictions made by both models.

In turn, for the models to make different predictions, they must make different assumptions about the prediction problem. More specifically, they must learn a different mapping function from inputs to outputs. We can achieve this in the simple case by training each model on a different sample of the training dataset, but there are many additional ways that we could achieve this difference; training different model types is one. These elements are how and ensemble methods work in the general sense, namely:

- **Members learn different mapping functions for the same problem**. This is to ensure that models make different prediction errors.
- **Predictions made by members are combined in some way**. This is to ensure that the differences in prediction errors are exploited.

We don't simply smooth out the prediction errors, although we can; instead, we smooth out the mapping function learned by the contributing members. The improved mapping function allows better predictions to be made. This is a deeper point and it is important that we understand it. Let's take a closer look at what it means for both classification and regression tasks.

3.3 Intuition for Classification Ensembles

Classification predictive modeling refers to problems where a class label must be predicted from examples of input. A model may predict a crisp class label, e.g. a categorical variable, or the probabilities for all possible categorical outcomes. In the simple case, the crisp class labels predicted by ensemble members can be combined by voting, e.g. the statistical mode or label with the most votes determines the ensemble prediction. Class probabilities predicted by ensemble members can be summed and normalized.

Functionally, some process like this is occurring in an ensemble for a classification task, but the effect is on the mapping function from input examples to class labels or probabilities. Let's stick with labels for now. The most common way to think about the mapping function for classification is by using a plot where the input data represents a point in an n-dimensional

space defined by the extent of the input variables, called the feature space. For example, if we had two input features, $x1$ and $x2$, both in the range zero to one, then the input feature space would be two-dimensional plane and each example in the dataset would be a point on that plane. Each point can then be assigned a color or shape based on the class label.

A model that learns how to classify points in effect draws lines in the feature space to separate examples. We can sample points in the feature space in a grid and get a map of how the model thinks the feature space should be by each class label. The separation of examples in the feature space by the model is called the decision boundary and a plot of the grid or map of how the model classifies points in the feature space is called a decision boundary plot. Now consider an ensemble where each model has a different mapping of inputs to outputs. In effect, each model has a different decision boundary or different idea of how to split up in the feature space by class label. Each model will draw the lines differently and make different errors.

When we combine the predictions from these multiple different models, we are in effect averaging the decision boundaries. We are defining a new decision boundary that attempts to learn from all the different views on the feature space learned by contributing members. The figure below taken from Page 1 of *Ensemble Machine Learning* provides a useful depiction of this.

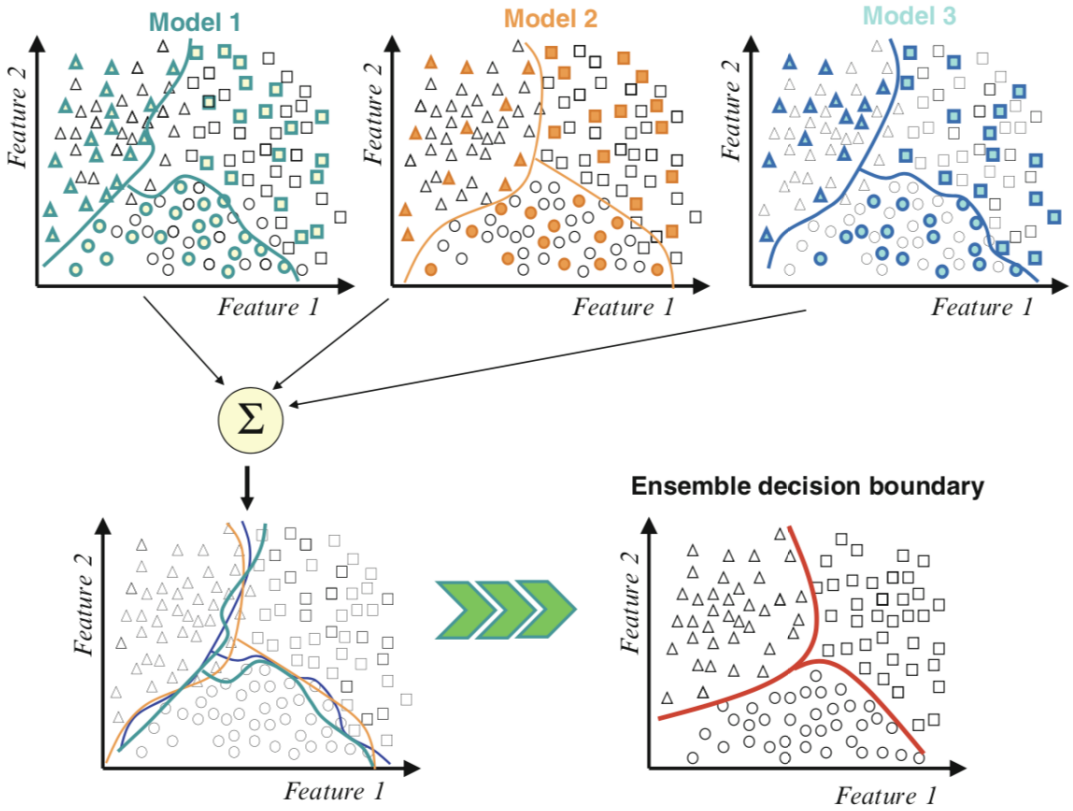

Figure 3.1: Example of Combining Decision Boundaries Using an Ensemble. Taken from *Ensemble Machine Learning*, 2012.

We can see the contributing members along the top, each with different decision boundaries in the feature space. Then the bottom-left draws all of the decision boundaries on the same plot

showing how they differ and make different errors. Finally, we can combine these boundaries to in-effect create a new generalized decision boundary in the bottom-right that better captures the true but unknown division of the feature space, resulting in better predictive performance.

3.4 Intuition for Regression Ensembles

Regression predictive modeling refers to problems where a numerical value must be predicted from examples of input. In the simple case, the numeric predictions made by ensemble members can be combined using statistical measures like the mean, although more complex combinations can be used. Like classification, the effect of the ensemble is that the mapping functions of each contributing member are averaged or combined. The most common way to think about the mapping function for regression is by using a line plot where the output variable is another dimension added to the input feature space. The relationship of the feature space and the target variable dimension can then be summarized as a hyperplane, e.g. a line in many dimensions.

This is mind-bending, so let's consider the simplest case where we have one numerical input and one numerical output. Consider a plane or graph where the x-axis represents the input feature and the y-axis represents the target variable. We can plot each example in the dataset as a point on this plot. A model that learns the mapping from input to outputs in effect learns a hyperplane that connects the points in the feature space to the target variable. We can sample a grid of points in the input feature space to devise values for the target variable and draw a line to connect them to represent this hyperplane. In our two-dimensional case, this is a line that passes through the points on the plot. Any point where the line does not pass through the plot represents a prediction error and the distance from the line to the point is the magnitude of the error.

Now consider an ensemble where each model has a different mapping of inputs to outputs. In effect, each model has a different hyperplane connecting the feature space to the target. Each model will draw different lines and make different errors with different magnitudes. When we combine the predictions from these multiple different models we are, in effect, averaging the hyperplanes. We are defining a new hyperplane that attempts to learn from all the different features on how to map inputs to outputs. The figure below gives an example of a one-dimensional input feature space and a target space with different learned hyperplane mappings.

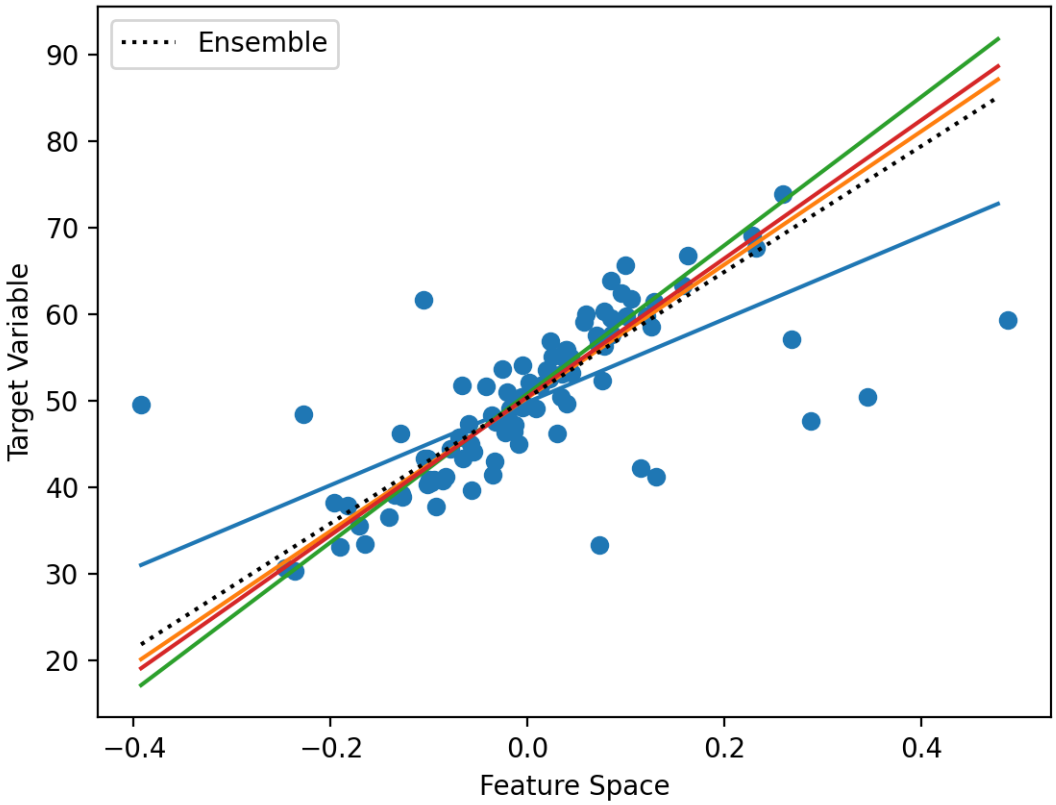

Figure 3.2: Example of Combining Hyperplanes Using an Ensemble

We can see the dots representing points from the training dataset. We can also see a number of different straight lines through the data. The models do not have to learn straight lines, but in this case, they have. Finally, we can see a dashed black line that shows the ensemble average of all of the models, resulting in lower prediction error.

3.5 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/3frGM1A

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Ensemble learning, Scholarpedia.
  http://www.scholarpedia.org/article/Ensemble_learning

3.6 Summary

In this tutorial, you discovered the intuition behind how ensemble learning methods work. Specifically, you learned:

- Ensemble learning methods work by in-effect combining the mapping functions learned by contributing members.
- Ensembles for classification are best understood by the combination of decision boundaries of members.
- Ensembles for regression are best understood by the combination of hyperplanes of members.

Next

This was the final tutorial in this part, in the next part we will take a closer look at the background required for understanding and using ensemble learning algorithms.

Part III

Background

Chapter 4

Ensemble Member Diversity

Ensemble learning combines the predictions from machine learning models for classification and regression. We use ensemble methods to achieve improved predictive performance, and it is this improvement over any of the contributing models that defines whether an ensemble is good or not. A property that is present in a good ensemble is the diversity of the predictions made by contributing models. Diversity is a slippery concept as it has not been precisely defined; nevertheless, it provides a useful practical heuristic for designing good ensemble models. In this tutorial, you will discover ensemble diversity in machine learning. After reading this tutorial, you will know:

- A good ensemble is one that has better performance than any contributing model.
- Ensemble diversity is a property of a good ensemble where contributing models make different errors with the same input.
- Seeking independent models and uncorrelated predictions provides a guide for thinking about and introducing diversity into ensemble models.

Let's get started.

4.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Makes a Good Ensemble
2. What Is Ensemble Diversity
3. Methods for Increasing Diversity

4.2 What Makes a Good Ensemble

An ensemble is a machine learning model that combines the predictions from multiple other models. This often has the effect of reducing prediction error and improving the generalization of the model. But this is not always the case. Sometimes the ensemble performs no better than

a well-performing contributing member to the ensemble. Even worse, sometimes an ensemble will perform worse than any contributing member. This raises the question as to what makes a good ensemble. A good ensemble is an ensemble that performs better than any contributing member. That is, it is a model that has lower prediction error for regression or higher accuracy for classification.

- **Good Ensemble**: A model that performs better than any single contributing model.

This can be evaluated empirically using a train and test set or a resampling technique like k-fold cross-validation. The results can similarly be estimated for each contributing model and the results compared directly to see if the definition of a *good ensemble* is met. What properties of a good ensemble differentiate it from other ensembles that perform as good or worse than any contributing member? This is a widely studied question with many ideas. The consistency is that a good ensemble has diversity.

4.3 What Is Ensemble Diversity

Ensemble diversity refers to differences in the decisions or predictions made by the ensemble members.

> Ensemble diversity, that is, the difference among the individual learners, is a fundamental issue in ensemble methods.
>
> — Page 99, *Ensemble Methods*, 2012.

Two ensemble members that make identical predictions are considered not diverse. Ensembles that make completely different predictions in all cases are maximally diverse, although this is most unlikely.

> Making different errors on any given sample is of paramount importance in ensemble-based systems. After all, if all ensemble members provide the same output, there is nothing to be gained from their combination.
>
> — Page 5, *Ensemble Machine Learning*, 2012.

Therefore, some level of diversity in predictions is desired or even required in order to construct a good ensemble. This is often simplified in discussion to the desire for diverse ensemble members, e.g. the models themselves, that in turn will produce diverse predictions, although diversity in predictions is truly what we seek. Ideally, diversity would mean that the predictions made by each ensemble member are independent and uncorrelated.

> ... classifier outputs should be independent or preferably negatively correlated.
>
> — Page 5, *Ensemble Machine Learning*, 2012.

Independence is a term from probability theory and refers to the case where one event does not influence the probability of the occurrence of another event. Events can influence each other in many different ways. One ensemble may influence another if it attempts to correct the predictions made by it. As such, depending on the ensemble type, models may be naturally dependent or independent.

- **Independence**: Whether the occurrence of an event affects the probability of subsequent events.

Correlation is a term from statistics and refers to two variables changing together. It is common to calculate a normalized correlation score between -1.0 and 1.0 where a score of 0.0 indicates no correlation, e.g. uncorrelated. A score of 1.0 or -1.0 one indicates perfect positive and negative correlation respectively and indicates two models that always predict the same (or inverse) outcome.

> ... if the learners are independent, i.e., [correlation] = 0, the ensemble will achieve a factor of T of error reduction than the individual learners; if the learners are totally correlated, i.e., [correlation] = 1, no gains can be obtained from the combination.
>
> — Page 99, *Ensemble Methods*, 2012.

In practice, ensembles often exhibit a weak or modest positive correlation in their predictions.

- **Correlation**: The degree to which variables change together.

As such, ensemble models are constructed from constituent models that may or may not have some dependency and some correlation between the decisions or predictions made. Constructing good ensembles is a challenging task. For example, combining a bunch of top-performing models will likely result in a poor ensemble as the predictions made by the models will be highly correlated. You might be better off combining the predictions from a few top-performing individual models with the prediction from a few weaker models, which is non-intuitive.

> So, it is desired that the individual learners should be accurate and diverse. Combining only accurate learners is often worse than combining some accurate ones together with some relatively weak ones, since complementarity is more important than pure accuracy.
>
> — Page 100, *Ensemble Methods*, 2012.

Sadly, there is no generally agreed upon measure of ensemble diversity and ideas of independence and correlation are only guides or proxies for thinking about the properties of good ensembles.

> Unfortunately, though diversity is crucial, we still do not have a clear understanding of diversity; for example, currently there is no well-accepted formal definition of diversity.
>
> — Page 100, *Ensemble Methods*, 2012.

Nevertheless, as guides, they are useful as we can devise techniques that attempt to reduce the correlation between the models.

4.4 Methods for Increasing Diversity

The objective for developing good ensembles is considered through the lens of increasing the diversity of ensemble members. Models in an ensemble may be more dependent if they share the same algorithm used to train the model and/or the same dataset used to train the model. Models in an ensemble may have higher correlation between their predictions for the same general reasons. Therefore, approaches that make the models and/or the training data more different are desirable.

> Diversity may be obtained through different presentations of the input data, as in bagging, variations in learner design, or by adding a penalty to the outputs to encourage diversity.
>
> — Page 93, *Pattern Classification Using Ensemble Methods*, 2010.

It can be helpful to have a framework for thinking about techniques for managing the diversity of ensembles, and there are many to choose from. For example, Zhi-Hua Zhou in Chapter 5 of his 2012 book titled *Ensemble Methods: Foundations and Algorithms* proposes a framework of four approaches to generating diversity. In summary, they are:

- **Data Sample Manipulation**: e.g. sample the training dataset differently for each model.
- **Input Feature Manipulation**: e.g. train each model on different groups of input features.
- **Learning Parameter Manipulation**: e.g. train models with different hyperparameter values.
- **Output Representation Manipulation**: e.g. train models with differently modified target values.

Perhaps the most common approaches are changes to the training data, that is data samples and input features, often combined in a single approach. For another example of a taxonomy for generating a diverse ensemble, Lior Rokach in Chapter 4 his 2010 book titled *Pattern Classification Using Ensemble Methods* suggests a similar taxonomy, summarized as:

- **Manipulating the Inducer**, e.g. manipulating how models are trained.
  - Vary hyperparameters.
  - Vary the starting point.
  - Vary optimization algorithm.
- **Manipulating the Training Sample**, e.g. manipulating the data used for training.
  - Resampling.
- **Changing the target attribute representation**, e.g. manipulating the target variable.
  - Vary encoding.

  - Error-correcting codes.
  - Label switching.

- **Partitioning the search space**, e.g. manipulating the number of input features.
  - Random subspace.
  - Feature selection.

- **Hybridization**, e.g. varying model types or a mixture of the above methods.

The hybridization approach is common both in popular ensemble methods that combine multiple approaches at generating diversity in a single algorithm, as well as simply varying the algorithms (model types) that comprise the ensemble.

> Bundling competing models into ensembles almost always improves generalization — and using different algorithms as the perturbation operator is an effective way to obtain the requisite diversity of components.
>
> — Page 89, *Ensemble Methods in Data Mining*, 2010.

Although we could investigate how to quantify ensemble diversity, effort may be bester spent on how to generate and manage the diversity of the ensemble. We can harness this knowledge in developing our own ensemble methods, first trying standard and well-understood ensemble learning methods, then tailoring them for our own specific datasets.

4.5 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5
- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/3frGM1A

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Ensemble learning, Scholarpedia.
  http://www.scholarpedia.org/article/Ensemble_learning

4.6 Summary

In this tutorial, you discovered ensemble diversity in machine learning. Specifically, you learned:

- A good ensemble is one that has better performance than any contributing model.
- Ensemble diversity is a property of a good ensemble where contributing models make different errors with the same input.
- Seeking independent models and uncorrelated predictions provides a guide for thinking about and introducing diversity into ensemble models.

Next

In the next section, we will take a closer look at the different techniques for combining predictions in ensemble learning algorithms.

Chapter 5

Techniques for Combining Predictions

Ensemble methods involve combining the predictions from multiple models. The combination of the predictions is a central part of the ensemble method and depends heavily on the types of models that contribute to the ensemble and the type of prediction problem that is being modeled, such as a classification or regression. Nevertheless, there are common or standard techniques that can be used to combine predictions that can be easily implemented and often result in good or best predictive performance. In this tutorial, you will discover common techniques for combining predictions for ensemble learning. After reading this tutorial, you will know:

- Combining predictions from contributing models is a key property of an ensemble model.
- Voting techniques are most commonly used when combining predictions for classification.
- Statistical techniques are most commonly used when combining predictions for regression.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Combining Predictions for Ensemble Learning
2. Combining Classification Predictions
3. Combining Regression Predictions

5.2 Combining Predictions for Ensemble Learning

A key part of an ensemble learning method involves combining the predictions from multiple models. It is through the combination of the predictions that the benefit of the ensemble learning method is achieved, namely better predictive performance. As such, there are many ways that predictions can be combined, so much so that it is an entire field of study.

> After generating a set of base learners, rather than trying to find the best single learner, ensemble methods resort to combination to achieve a strong generalization ability, where the combination method plays a crucial role.

— Page 67, *Ensemble Methods*, 2012.

Standard ensemble machine learning algorithms do prescribe how to combine predictions; nevertheless, it is important to consider the topic in isolation for a number of reasons, such as:

- Interpreting the predictions made by standard ensemble algorithms.
- Manually specifying a custom prediction combination method for an algorithm.
- Developing your own ensemble methods.

Ensemble learning methods are typically not very complex and developing your own ensemble method or specifying the manner in which predictions are combined is relatively easy and common practice. The way that predictions are combined depends on the models that are making predictions and the type of prediction problem.

> The strategy used in this step depends, in part, on the type of classifiers used as ensemble members. For example, some classifiers, such as support vector machines, provide only discrete-valued label outputs.

— Page 6, *Ensemble Machine Learning*, 2012.

For example, the form of the predictions made by the models will match the type of prediction problem, such as regression for predicting numbers and classification for predicting class labels. Additionally, some model types may be only able to predict a class label or class probability distribution, whereas others may be able to support both for a classification task. We will use this division of prediction type based on problem type as the basis for exploring the common techniques used to combine predictions from contributing models in an ensemble. In the next section, we will take a look at how to combine predictions for classification predictive modeling tasks.

5.3 Combining Classification Predictions

Classification refers to predictive modeling problems that involve predicting a class label given an input. The prediction made by a model may be a crisp class label directly or may be a probability that an example belongs to each class, referred to as the probability of class membership. The performance of a classification problem is often measured using accuracy or a related count or ratio of correct predictions. In the case of evaluating predicted probabilities, they may be converted to crisp class labels by selecting a cut-off threshold, or evaluated using specialized metrics such as cross-entropy. We will review combining predictions for classification separately for both class labels and probabilities.

5.3.1 Combining Predicted Class Labels

A predicted class label is often mapped to something meaningful in the problem domain. For example, a model may predict a color such as *red*, *green*, or *blue*. Internally though, the model predicts a numerical representation for the class label such as 0 for *red*, 1 for *green*, and 2 for *blue* for our color classification example. Methods for combining class labels are perhaps easier to consider if we work with the integer encoded class labels directly. Perhaps the simplest, most common, and often most effective approach is to combine the predictions by voting.

> Voting is the most popular and fundamental combination method for nominal outputs.
>
> — Page 71, *Ensemble Methods*, 2012.

Voting generally involves each model that makes a prediction assigning a vote for the class that was predicted. The votes are tallied and an outcome is then chosen using the votes or tallies in some way. There are many types of voting, so let's look at the four most common:

- Plurality Voting.
- Majority Voting.
- Unanimous Voting.
- Weighted Voting.

Simple voting, called plurality voting, selects the class label with the most votes. If two or more classes have the same number of votes, then the tie is broken arbitrarily, although in a consistent manner, such as sorting the class labels that have a tie and selecting the first, instead of selecting one randomly. This is important so that the same model with the same data always makes the same prediction. Given ties, it is common to have an odd number of ensemble members in an attempt to automatically break ties, as opposed to an even number of ensemble members where ties may be more likely. From a statistical perspective, this is called the mode or the most common value from the collection of predictions. For example, consider the three predictions made by a model for a three-class color prediction problem:

- Model 1 predicts *green* or 1.
- Model 2 predicts *green* or 1.
- Model 3 predicts *red* or 0.

The votes are, therefore:

- Red Votes: 1
- Green Votes: 2
- Blue Votes: 0

The prediction would be *green* given it has the most votes. Majority voting selects the class label that has more than half the votes. If no class has more than half the votes, then a *no prediction* is made. Interestingly, majority voting can be proven to be an optimal method for combining classifiers, if they are independent.

> If the classifier outputs are independent, then it can be shown that majority voting is the optimal combination rule.
>
> — Page 1, *Ensemble Machine Learning*, 2012.

Unanimous voting is related to majority voting in that instead of requiring half the vote (a majority), the method requires all models to predict the same value, otherwise, no prediction is made (e.g. a null or no decision). Weighted voting weighs the prediction made by each model in some way. One example would be to weigh predictions based on the average performance of the model, such as classification accuracy.

> The weight of each classifier can be set proportional to its accuracy performance on a validation set.
>
> — Page 67, *Pattern Classification Using Ensemble Methods*, 2010.

Assigning weights to classifiers can become a project in and of itself and could involve using an optimization algorithm and a holdout dataset, a linear model, or even another machine learning model entirely.

> So, how do we assign the weights? If we knew, a priori, which classifiers would work better, we would only use those classifiers. In the absence of such information, a plausible and commonly used strategy is to use the performance of a classifier on a separate validation (or even training) dataset, as an estimate of that classifier's generalization performance.
>
> — Page 8, *Ensemble Machine Learning*, 2012.

The idea of weighted voting is that some classifiers are more likely to be accurate than others and we should reward them by giving them a larger share of the votes.

> If we have reason to believe that some of the classifiers are more likely to be correct than others, weighting the decisions of those classifiers more heavily can further improve the overall performance compared to that of plurality voting.
>
> — Page 7, *Ensemble Machine Learning*, 2012.

5.3.2 Combining Predicted Class Probabilities

Probabilities summarize the likelihood of an event as a numerical value between 0.0 and 1.0. When predicted for class membership, it involves a probability assigned for each class, together summing to the value 1.0; for example, a model may predict:

- Red: 0.75
- Green: 0.10
- Blue: 0.15

We can see that class *red* has the highest probability or is the most likely outcome predicted by the model and that the distribution of probabilities across the classes (0.75 + 0.10 + 0.15) sum to 1.0. The way that the probabilities are combined depends on the outcome that is required. For example, if probabilities are required, then the independent predicted probabilities can be combined directly. Perhaps the simplest approach for combining probabilities is to sum the probabilities for each class and pass the predicted values through a softmax function (e.g. convert arbitrary output values into a consistent multinomial probability disquisition). This ensures that the scores are appropriately normalized, meaning the probabilities across the class labels sum to 1.0.

> ... such outputs — upon proper normalization (such as softmax normalization [...]) — can be interpreted as the degree of support given to that class
>
> — Page 8, *Ensemble Machine Learning*, 2012.

More commonly we wish to predict a class label from predicted probabilities. The most common approach is to use voting, where the predicted probabilities represent the vote made by each model for each class. Votes are then summed and a voting method from the previous section can be used, such as selecting the label with the largest summed probabilities or the largest mean probability.

- Vote Using Mean Probabilities
- Vote Using Sum Probabilities
- Vote Using Weighted Sum Probabilities

Generally, this approach to treating probabilities as votes for choosing a class label is referred to as soft voting.

> If all the individual classifiers are treated equally, the simple soft voting method generates the combined output by simply averaging all the individual outputs ...
>
> — Page 76, *Ensemble Methods*, 2012.

5.4 Combining Regression Predictions

Regression refers to predictive modeling problems that involve predicting a numeric value given an input. The performance for a regression problem is often measured using average error, such as mean absolute error or root mean squared error. Combining numerical predictions often involves using simple statistical methods; for example:

- Mean Predicted Value
- Median Predicted Value

Both give the central tendency of the distribution of predictions.

> Averaging is the most popular and fundamental combination method for numeric outputs.
>
> — Page 68, *Ensemble Methods*, 2012.

The mean, also called the average, is the normalized sum of the predictions. The Mean Predicted Value is more appropriate when the distribution of predictions is Gaussian or nearly Gaussian. For example, the mean is calculated as the sum of predicted values divided by the total number of predictions. If three models predicted the following values:

- Model 1: 99.00

- Model 2: 101.00
- Model 3: 98.00

The mean predicted would be calculated as:

$$\begin{aligned} \text{Mean Prediction} &= \frac{99.00 + 101.00 + 98.00}{3} \\ &= \frac{298.00}{3} \\ &= 99.33 \end{aligned} \tag{5.1}$$

> Owing to its simplicity and effectiveness, simple averaging is among the most popularly used methods and represents the first choice in many real applications.
>
> — Page 69, *Ensemble Methods*, 2012.

The median is the middle value if all predictions were ordered and is also referred to as the fiftieth percentile. The Median Predicted Value is more appropriate to use when the distribution of predictions is not known or does not follow a Gaussian probability distribution. Depending on the nature of the prediction problem, a conservative prediction may be desired, such as the maximum or the minimum. Additionally, the distribution can be summarized to give a measure of uncertainty, such as reporting three values for each prediction:

- Minimum Predicted Value
- Median Predicted Value
- Maximum Predicted Value

As with classification, the predictions made by each model can be weighted by expected model performance or some other value, and the weighted mean of the predictions can be reported.

5.5 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5
- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/3frGM1A

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Ensemble learning, Scholarpedia.
  http://www.scholarpedia.org/article/Ensemble_learning

5.6 Summary

In this tutorial, you discovered common techniques for combining predictions for ensemble learning. Specifically, you learned:

- Combining predictions from contributing models is a key property of an ensemble model.
- Voting techniques are most commonly used when combining predictions for classification.
- Statistical techniques are most commonly used when combining predictions for regression.

Next

In the next section, we will take a closer look at the complexity of ensemble learning algorithms and why they seem to violate Occam's razor.

Chapter 6

Ensemble Complexity

Occam's razor suggests that in machine learning, we should prefer simpler models with fewer coefficients over complex models like ensembles. Taken at face value, the razor is a heuristic that suggests more complex hypotheses make more assumptions that, in turn, will make them too narrow and not generalize well. In machine learning, it suggests complex models like ensembles will overfit the training dataset and perform poorly on new data. In practice, ensembles are almost universally the type of model chosen on projects where predictive skill is the most important consideration. Further, empirical results show a continued reduction in generalization error as the complexity of an ensemble learning model is incrementally increased. These findings are at odds with the Occam's razor principle taken at face value. In this tutorial, you will discover how to reconcile Occam's Razor with ensemble machine learning. After completing this tutorial, you will know:

- Occam's razor is a heuristic that suggests choosing simpler machine learning models as they are expected to generalize better.
- The heuristic can be divided into two razors, one of which is true and remains a useful tool and the other that is false and should be abandoned.
- Ensemble learning algorithms like boosting provide a specific case of how the second razor fails and added complexity can result in lower generalization error.

Let's get started.

6.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Occam's Razor for Model Selection
2. Occam's Two Razors for Machine Learning
3. Occam's Razor and Ensemble Learning

6.2 Occam's Razor for Model Selection

Model selection is the process of choosing one from among possibly many candidate machine learning models for a predictive modeling project. It is often straightforward to select a model based on its expected performance, e.g. choose the model with the highest accuracy or lowest prediction error. Another important consideration is to choose simpler models over complex models. Simpler models are typically defined as models that make fewer assumptions or have fewer elements, most commonly characterized as fewer coefficients (e.g. rules, layers, weights, etc.). The rationale for choosing simpler models is tied back to Occam's Razor.

> The idea is that the best scientific theory is the smallest one that explains all the facts.
>
> — Page 197, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

Occam's Razor is an approach to problem-solving and is commonly invoked to mean that if all else is equal, we should prefer the simpler solutions.

- **Occam's Razor**: If all else is equal, the simplest solution is correct.

It is named for William of Ockham and was proposed to counter ever more elaborate philosophy without equivalent increases in predictive power.

> William of Occam's famous razor states that "Nunquam ponenda est pluralitas sin necesitate", which, approximately translated, means "Entities should not be multiplied beyond necessity".
>
> — *Occam's Two Razors: The Sharp and the Blunt*, 1998.

It is not a rule, more of a heuristic for problem-solving, and is commonly invoked in science to prefer simpler hypotheses that make fewer assumptions over more complex hypotheses that make more assumptions.

> There is a long-standing tradition in science that, other things being equal, simple theories are preferable to complex ones. This is known as Occam's Razor after the medieval philosopher William of Occam (or Ockham).
>
> — Page 197, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

The problem with complex hypotheses with more assumptions is that they are likely too specific. They may include details of specific cases that are at hand or easily available, and in turn, may not generalize to new cases. That is, the more assumptions a hypothesis has, the more narrow it is expected to be in its application. Conversely, fewer assumptions suggests a more general hypothesis with greater predictive power to more cases.

- **Simple Hypothesis**: Fewer assumptions, and in turn, broad applicability.
- **Complex Hypothesis**: More assumptions, and in turn, narrow applicability.

This has implications in machine learning, as we are specifically trying to generalize to new unseen cases from specific observations, referred to as inductive reasoning. If Occam's Razor suggests that more complex models don't generalize well, then in applied machine learning, it suggests we should choose simpler models as they will have lower prediction errors on new data. **If this is true, then how can we justify using an ensemble machine learning algorithm?**

By definition, ensemble machine learning algorithms are more complex than a single machine learning model, as they are composed of many individual machine learning models. Occam's razor suggests that the added complexity of ensemble learning algorithms means that they will not generalize as well as simpler models fit on the same dataset. Yet ensemble machine learning algorithms are the dominant solution when predictive skill on new data is the most important concern, such as machine learning competitions. Ensembles have been studied at great length and have been shown not to overfit the training dataset in study after study.

> It has been empirically observed that certain ensemble techniques often do not overfit the model, even when the ensemble contains thousands of classifiers.
>
> — Page 40, *Pattern Classification Using Ensemble Methods*, 2010.

How can this inconsistency be reconciled?

6.3 Occam's Two Razors for Machine Learning

The conflict between the expectation of simpler models generalizing better in theory and complex models like ensembles generalizing better in practice was mostly ignored as an inconvenient empirical finding for a long time. In the late 1990s, the problem was specifically studied by Pedro Domingos and published in the award-winning 1996 paper titled *Occam's Two Razors: The Sharp and the Blunt*, and follow-up 1999 journal article *The Role of Occam's Razor in Knowledge Discovery*. In the work, Domingos defines the problem as two specific commonly asserted implications of Occam's Razor in applied machine learning, which he refers to as *Occam's Two Razors* in machine learning, they are (taken from the paper):

- **First razor**: Given two models with the same generalization error, the simpler one should be preferred because simplicity is desirable in itself.
- **Second razor**: Given two models with the same training-set error, the simpler one should be preferred because it is likely to have lower generalization error.

Domingos then enumerates a vast number of examples for and against each razor from both theory and empirical studies in machine learning. The **first razor** suggests if two models have the same expected performance on data not seen during training, we should prefer the simpler model. Domingos highlights that this razor holds and provides a good heuristic on machine learning projects. The **second razor** suggests that if two models have the same performance on a training dataset, then the simpler model should be chosen because it is expected to generalize better when used to make predictions on new data.

This seems sensible on the surface. It is the argument behind not adopting ensemble algorithms in a machine learning project because they are very complex compared to other models and expected to not generalize. It turns out that this razor cannot be supported by the evidence from the machine learning literature.

> All of this evidence points to the conclusion that not only is the second razor not true in general; it is also typically false in the types of domains KDD has been applied to.
>
> — *Occam's Two Razors: The Sharp and the Blunt*, 1998.

6.4 Occam's Razor and Ensemble Learning

The finding begins to sound intuitive once you mull on it for a while. For example, in practice, we would not choose a machine learning model based on its performance on the training dataset alone. We intuitively, or perhaps after a lot of experience, tacitly expect the estimate of performance on a training set to be a poor estimate of performance on a holdout dataset.

We have this expectation because the model can overfit the training dataset. Yet, less intuitively, overfitting the training dataset can lead to better performance on a holdout test set. This has been observed many times in practice in systematic studies of ensemble learning algorithms. A common situation involves plotting the performance of a model on the training dataset and a holdout test dataset each iteration of learning for the model, such as training epochs or iterations for models that support incremental learning.

If learning on the training dataset is set up to continue for a large number of training iterations and the curves observed, it can often be seen that performance on the training dataset will fall to zero error. This is to be expected as we might think that the model will overfit the training dataset given enough resources and time to train. Yet the performance on the test set will continue to improve, even while the performance on the training set remains fixed at zero error.

> ... occasionally, the generalization error would continue to improve long after the training error had reached zero.
>
> — Page 40, *Ensemble Methods in Data Mining*, 2010.

This behavior can be observed with ensemble learning algorithms like boosting and bagging, where performance on the holdout dataset will continue to improve as additional model members are added to the ensemble.

> One very surprising finding is that performing more boosting iterations can reduce the error on new data long after the classification error of the combined classifier on the training data has dropped to zero.
>
> — Page 489, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

That is, the model complexity is incrementally increased, which systematically decreases the error on unseen data, e.g. generalization error. The additional training cannot improve the performance on the training dataset; it has no possible improvement to make.

> Performing more boosting iterations without reducing training error does not explain the training data any better, and it certainly adds complexity to the combined classifier.

> — Page 490, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

This finding directly contradicts the second razor and supports Domingos' argument about abandoning the second razor.

> The first one is largely uncontroversial, while the second one, taken literally, is false.
>
> — *Occam's Two Razors: The Sharp and the Blunt*, 1998.

This problem has been studied and can generally be explained by the ensemble algorithms learning to be more confident in their predictions on the training dataset, which carry over to the hold out data.

> The contradiction can be resolved by considering the classifier's confidence in its predictions.
>
> — Page 490, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

The first razor remains an important heuristic in applied machine learning. The key aspect of this razor is the predicate of *all else being equal*. That is, if two models are compared, they must be compared using their generalization error on a holdout dataset or estimated using k-fold cross-validation. If their performance is equal under these circumstances, then the razor can come into effect and we can choose the simpler solution.

This is not the only way to choose models. We may choose a simpler model because it is easier to interpret, and this remains valid if model interpretability is a more important project requirement than predictive skill. Ensemble learning algorithms are unambiguously a more complex type of model when the number of model parameters is considered the measure of complexity. As such, an open problem in machine learning involves alternate measures of complexity.

6.5 Further Reading

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

Papers

- *Occam's Two Razors: The Sharp and the Blunt*, 1998.
  https://www.aaai.org/Library/KDD/1998/kdd98-006.php
- *The Role of Occam's Razor in Knowledge Discovery*, 1999.
  https://link.springer.com/article/10.1023/A:1009868929893

Books

- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/2QCEC5q
- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2tlRP9V

Articles

- Occam's razor, Wikipedia.
  https://en.wikipedia.org/wiki/Occam%27s_razor
- William of Ockham, Wikipedia.
  https://en.wikipedia.org/wiki/William_of_Ockham

6.6 Summary

In this tutorial, you discovered how to reconcile Occam's Razor with ensemble machine learning. Specifically, you learned:

- Occam's razor is a heuristic that suggests choosing simpler machine learning models as they are expected to generalize better.
- The heuristic can be divided into two razors, one of which is true and remains a useful tool, and the other that is false and should be abandoned.
- Ensemble learning algorithms like boosting provide a specific case of how the second razor fails and added complexity can result in lower generalization error.

Next

In the next section, we will take a closer look at the three main types or classes of ensemble learning algorithms in machine learning.

Chapter 7

Types of Ensemble Algorithms

Ensemble learning is a general meta approach to machine learning that seeks better predictive performance by combining the predictions from multiple models. Although there are a seemingly unlimited number of ensembles that you can develop for your predictive modeling problem, there are three methods that dominate the field of ensemble learning. So much so, that rather than algorithms per se, each is a field of study that has spawned many more specialized methods.

The three main classes of ensemble learning methods are bagging, boosting, and stacking, and it is important to both have a detailed understanding of each method and to consider them on your predictive modeling project. But, before that, you need a gentle introduction to these approaches and the key ideas behind each method prior to layering on math and code. In this tutorial, you will discover the three standard ensemble learning techniques for machine learning. After completing this tutorial, you will know:

- Bagging involves fitting many decision trees on different samples of the same dataset and averaging the predictions.
- Boosting involves adding ensemble members sequentially that correct the prediction errors made by prior models and outputs a weighted average of the predictions.
- Stacking involves fitting many different model types on the same data and using another model to learn how to best combine the predictions.

Let's get started.

7.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Standard Ensemble Learning Strategies
2. Bagging Ensemble Learning
3. Boosting Ensemble Learning
4. Stacking Ensemble Learning

7.2 Standard Ensemble Learning Strategies

Ensemble learning refers to algorithms that combine the predictions from two or more models. Although there is nearly an unlimited number of ways that this can be achieved, there are perhaps three classes of ensemble learning techniques that are most commonly discussed and used in practice. Their popularity is due in large part to their ease of implementation and success on a wide range of predictive modeling problems.

> A rich collection of ensemble-based classifiers have been developed over the last several years. However, many of these are some variation of the select few well-established algorithms whose capabilities have also been extensively tested and widely reported.
>
> — Page 11, *Ensemble Machine Learning*, 2012.

Given their wide use, we can refer to them as *standard* ensemble learning strategies; they are:

- Bagging.
- Boosting.
- Stacking.

There is an algorithm that describes each approach, although more importantly, the success of each approach has spawned a myriad of extensions and related techniques. As such, it is more useful to describe each as a class of techniques or standard approaches to ensemble learning. Rather than dive into the specifics of each method, it is useful to step through, summarize, and contrast each approach. It is also important to remember that although discussion and use of these methods are pervasive, these three methods alone do not define the extent of ensemble learning. Next, let's take a closer look at bagging.

7.3 Bagging Ensemble Learning

Bootstrap aggregation, or bagging for short, is an ensemble learning method that seeks a diverse group of ensemble members by varying the training data.

> The name Bagging came from the abbreviation of Bootstrap AGGregatING. As the name implies, the two key ingredients of Bagging are bootstrap and aggregation.
>
> — Page 48, *Ensemble Methods*, 2012.

This typically involves using a single machine learning algorithm, almost always an unpruned decision tree, and training each model on a different sample of the same training dataset. The predictions made by the ensemble members are then combined using simple statistics, such as voting or averaging.

> The diversity in the ensemble is ensured by the variations within the bootstrapped replicas on which each classifier is trained, as well as by using a relatively weak classifier whose decision boundaries measurably vary with respect to relatively small perturbations in the training data.
>
> — Page 11, *Ensemble Machine Learning*, 2012.

Key to the method is the manner in which each sample of the dataset is prepared to train ensemble members. Each model gets its own unique sample of the dataset. Examples (rows) are drawn from the dataset at random, although with replacement.

> Bagging adopts the bootstrap distribution for generating different base learners. In other words, it applies bootstrap sampling to obtain the data subsets for training the base learners.
>
> — Page 48, *Ensemble Methods*, 2012.

Replacement means that if a row is selected, it is returned to the training dataset for potential re-selection in the same training sample. This means that a row of data may be selected zero, one, or multiple times for a given training dataset. This is called a bootstrap sample. It is a technique often used in statistics with small datasets to estimate the statistical value of a data sample. By preparing multiple different bootstrap samples and estimating a statistical quantity and calculating the mean of the estimates, a better overall estimate of the desired quantity relative to the population can be achieved than simply estimating from the dataset directly.

In the same manner, multiple different training datasets can be prepared, used to estimate a predictive model, and make predictions. Averaging the predictions across the models typically results in better predictions than a single model fit on the training dataset directly. We can summarize the key elements of bagging as follows:

- Bootstrap samples of the training dataset.
- Unpruned decision trees fit on each sample.
- Simple voting or averaging of predictions.

In summary, the contribution of bagging is in the varying of the training data used to fit each ensemble member, which, in turn, results in skillful but different models.

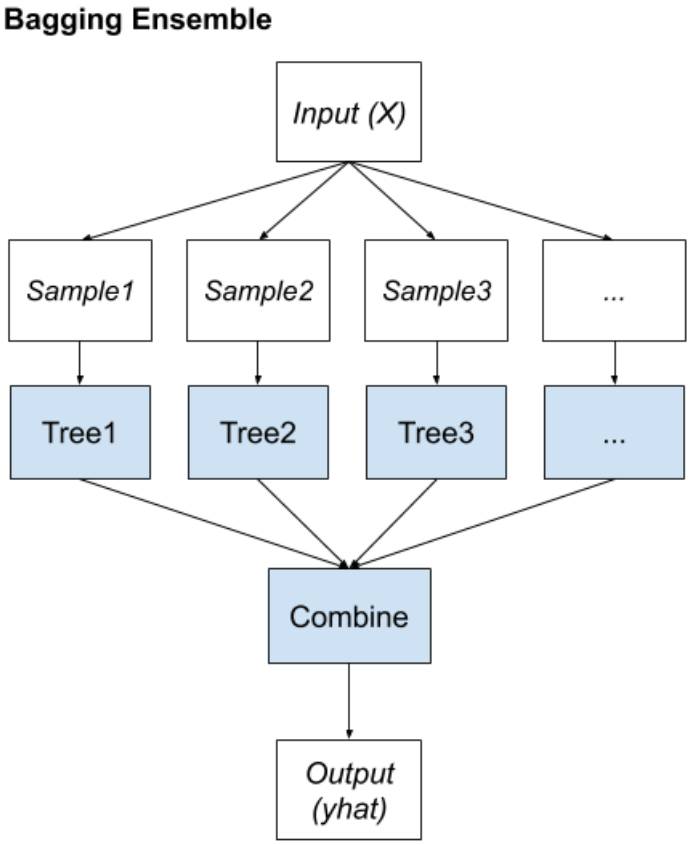

Figure 7.1: Example of a Bagging Ensemble.

Bagging is a general approach and easily extended. For example, more changes to the training dataset can be introduced, the algorithm fit on the training data can be replaced, and the mechanism used to combine predictions can be modified. Many popular ensemble algorithms are based on this approach, including:

- Bagged Decision Trees (canonical bagging)
- Random Forest
- Extra Trees

We will be taking a closer look bagging ensembles in *Part V* of this book. Next, let's take a closer look at boosting.

7.4 Boosting Ensemble Learning

Boosting is an ensemble method that seeks to change the training data to focus attention on examples that previous fit models on the training dataset have gotten wrong.

> In boosting, [...] the training dataset for each subsequent classifier increasingly focuses on instances misclassified by previously generated classifiers.

— Page 13, *Ensemble Machine Learning*, 2012.

The key property of boosting ensembles is the idea of correcting prediction errors. The models are fit and added to the ensemble sequentially such that the second model attempts to correct the predictions of the first model, the third corrects the second model, and so on. This typically involves the use of very simple decision trees that only make a single or a few decisions, referred to in boosting as weak learners. The predictions of the weak learners are combined using simple voting or averaging, although the contributions are weighed proportional to their performance or capability. The objective is to develop a so-called *strong-learner* from many purpose-built *weak-learners*.

> ... an iterative approach for generating a strong classifier, one that is capable of achieving arbitrarily low training error, from an ensemble of weak classifiers, each of which can barely do better than random guessing.

— Page 13, *Ensemble Machine Learning*, 2012.

Typically, the training dataset is left unchanged and instead, the learning algorithm is modified to pay more or less attention to specific examples (rows of data) based on whether they have been predicted correctly or incorrectly by previously added ensemble members. For example, the rows of data can be weighed to indicate the amount of focus a learning algorithm must give while learning the model. We can summarize the key elements of boosting as follows:

- Bias training data toward those examples that are hard to predict.
- Iteratively add ensemble members to correct predictions of prior models.
- Combine predictions using a weighted average of models.

The idea of combining many weak learners into strong learners was first proposed theoretically and many algorithms were proposed with little success. It was not until the Adaptive Boosting (AdaBoost) algorithm was developed that boosting was demonstrated as an effective ensemble method.

> The term boosting refers to a family of algorithms that are able to convert weak learners to strong learners.

— Page 23, *Ensemble Methods*, 2012.

Since AdaBoost, many boosting methods have been developed and some, like stochastic gradient boosting, may be among the most effective techniques for classification and regression on tabular (structured) data.

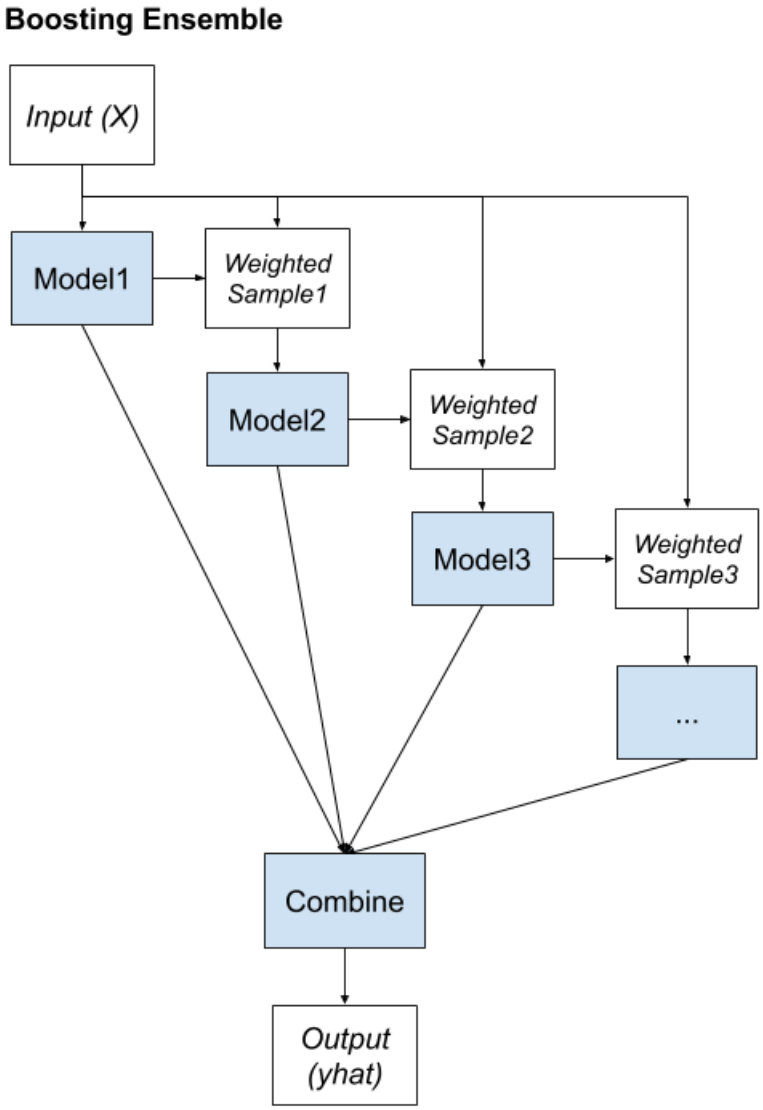

Figure 7.2: Example of a Boosting Ensemble.

To summarize, many popular ensemble algorithms are based on this approach, including:

- AdaBoost (canonical boosting)
- Gradient Boosting Machines
- Stochastic Gradient Boosting (XGBoost and similar)

We will be taking a closer look boosting ensembles in *Part VI* of this book. Next, let's take a closer look at stacking.

7.5 Stacking Ensemble Learning

Stacked Generalization, or stacking for short, is an ensemble method that seeks a diverse group of members by varying the model types fit on the training data and using a model to combine

predictions.

> Stacking is a general procedure where a learner is trained to combine the individual learners. Here, the individual learners are called the first-level learners, while the combiner is called the second-level learner, or meta-learner.
>
> — Page 83, *Ensemble Methods*, 2012.

Stacking has its own nomenclature where ensemble members are referred to as level-0 models and the model that is used to combine the predictions is referred to as a level-1 model. The two-level hierarchy of models is the most common approach, although more layers of models can be used. For example, instead of a single level-1 model, we might have 3 or 5 level-1 models and a single level-2 model that combines the predictions of level-1 models in order to make a prediction.

> Stacking is probably the most-popular meta-learning technique. By using a meta-learner, this method tries to induce which classifiers are reliable and which are not.
>
> — Page 82, *Pattern Classification Using Ensemble Methods*, 2010.

Any machine learning model can be used to aggregate the predictions, although it is common to use a linear model, such as linear regression for regression and logistic regression for binary classification. This encourages the complexity of the model to reside at the lower-level ensemble member models and simple models to learn how to harness the variety of predictions made.

> Using trainable combiners, it is possible to determine which classifiers are likely to be successful in which part of the feature space and combine them accordingly.
>
> — Page 15, *Ensemble Machine Learning*, 2012.

We can summarize the key elements of stacking as follows:

- Unchanged training dataset.
- Different machine learning algorithms for each ensemble member.
- Machine learning model to learn how to best combine predictions.

Diversity comes from the different machine learning models used as ensemble members. As such, it is desirable to use a suite of models that are learned or constructed in very different ways, ensuring that they make different assumptions and, in turn, have less correlated prediction errors.

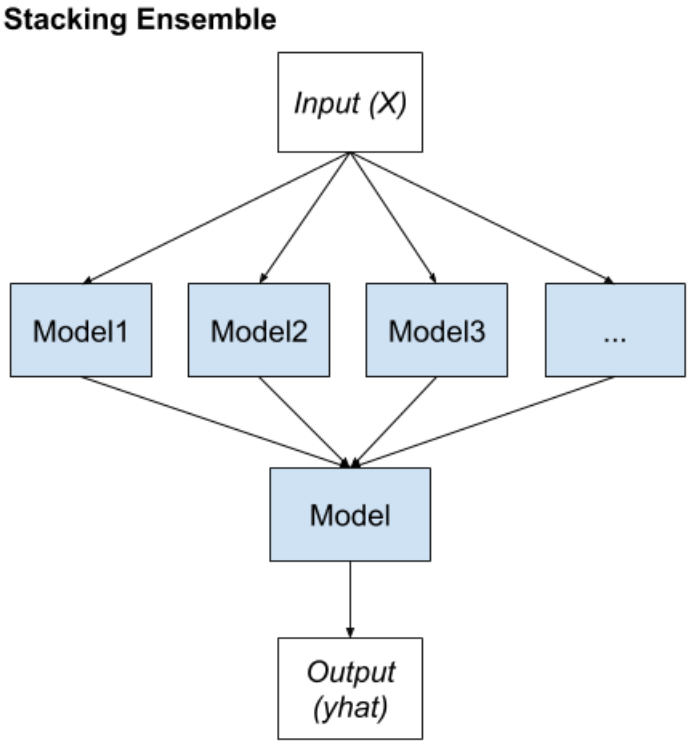

Figure 7.3: Example of a Stacking Ensemble.

Many popular ensemble algorithms are based on this approach, including:

- Stacked Generalization (canonical stacking)
- Blending Ensemble
- Super Learner Ensemble

We will be taking a closer look stacking ensembles in *Part VII* of this book. This completes our tour of the standard ensemble learning techniques.

7.6 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/3frGM1A

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Bootstrap aggregating, Wikipedia.
  https://en.wikipedia.org/wiki/Bootstrap_aggregating
- Boosting (machine learning), Wikipedia.
  https://en.wikipedia.org/wiki/Boosting_(machine_learning)

7.7 Summary

In this tutorial, you discovered the three standard ensemble learning techniques for machine learning. Specifically, you learned:

- Bagging involves fitting many decision trees on different samples of the same dataset and averaging the predictions.
- Boosting involves adding ensemble members sequentially that correct the prediction errors made by prior models and outputs a weighted average of the predictions.
- Stacking involves fitting many different model types on the same data and using another model to learn how to best combine the predictions.

Next

This was the final tutorial in this part, in the next part we will take a closer look at multiple model machine learning algorithms.

Part IV

Multiple Models

Chapter 8

One-vs-Rest and One-vs-One Models

Not all classification predictive models support multiclass classification. Algorithms such as the Perceptron, Logistic Regression, and Support Vector Machines were designed for binary classification and do not natively support classification tasks with more than two classes. One approach for using binary classification algorithms for multiple class classification problems is to split the multiclass classification dataset into multiple binary classification datasets and fit a binary classification model on each. Two different examples of this approach are the One-vs-Rest and One-vs-One strategies. Both of these approaches can be thought of as primitive types of ensemble algorithms that make use of multiple separate models in concert to make predictions. In this tutorial, you will discover One-vs-Rest and One-vs-One strategies for multiclass classification. After completing this tutorial, you will know:

- Binary classification models like logistic regression and SVM do not support multiclass classification natively and require meta-strategies.
- The One-vs-Rest strategy splits a multiclass classification into one binary classification problem per class.
- The One-vs-One strategy splits a multiclass classification into one binary classification problem per each pair of classes.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Binary Classifiers for Multiclass Classification
2. One-Vs-Rest for Multiclass Classification
3. One-Vs-One for Multiclass Classification

8.2 Binary Classifiers for Multiclass Classification

Classification is a predictive modeling problem that involves assigning a class label to an example. Binary classification are those tasks where examples are assigned exactly one of two classes. Multiclass classification is those tasks where examples are assigned exactly one of more than two classes.

- **Binary Classification:** Classification tasks with two classes.
- **Multiclass Classification**: Classification tasks with more than two classes.

Some algorithms are designed for binary classification problems. Examples include:

- Logistic Regression
- Perceptron
- Support Vector Machines

As such, they cannot be used for multiclass classification tasks, at least not directly. Instead, heuristic methods can be used to split a multiclass classification problem into multiple binary classification datasets and train a binary classification model each. Two examples of these heuristic methods include:

- One-vs-Rest (OvR)
- One-vs-One (OvO)

Let's take a closer look at each.

8.3 One-Vs-Rest for Multiclass Classification

One-vs-Rest (OvR for short, also referred to as One-vs-All or OvA) is a heuristic method for using binary classification algorithms for multiclass classification. It involves splitting the multiclass dataset into multiple binary classification problems. A binary classifier is then trained on each binary classification problem and predictions are made using the model that is the most confident. For example, given a multiclass classification problem with examples for each class *red*, *blue*, and *green*. This could be divided into three binary classification datasets as follows:

- Binary Classification Problem 1: red vs [blue, green]
- Binary Classification Problem 2: blue vs [red, green]
- Binary Classification Problem 3: green vs [red, blue]

A possible downside of this approach is that it requires one model to be created for each class. For example, three classes requires three models. This could be an issue for large datasets (e.g. millions of rows), slow models (e.g. SVM), or very large numbers of classes (e.g. hundreds of classes).

> The obvious approach is to use a one-versus-the-rest approach (also called one-vs-all), in which we train C binary classifiers, $fc(x)$, where the data from class c is treated as positive, and the data from all the other classes is treated as negative.
>
> — Page 503, *Machine Learning: A Probabilistic Perspective*, 2012.

This approach requires that each model predicts a class membership probability or a probability-like score. The argmax of these scores (class index with the largest score) is then used to predict a class. This approach is commonly used for algorithms that naturally predict numerical class membership probability or score, such as:

- Logistic Regression
- Perceptron

As such, the implementation of these algorithms in the scikit-learn library implements the OvR strategy by default when using these algorithms for multiclass classification. First, let's define a synthetic multiclass classification dataset that we can use as the basis of the exploration. We can use the `make_classification()` function to define a multiclass classification problem with 1,000 examples, 10 input features, and three classes. The example below demonstrates how to create the dataset and summarize the number of rows, columns, and classes in the dataset.

```
# multi-class classification dataset
from sklearn.datasets import make_classification
from collections import Counter
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    n_classes=3, random_state=1)
# summarize the dataset
print(X.shape, y.shape)
# summarize the number of examples in each class
print(Counter(y))
```

Listing 8.1: Example of defining a synthetic multiclass classification dataset.

Running the example creates the dataset and reports the number of rows and columns, confirming the dataset was created as expected with 1,000 examples and 10 input features. The number of examples in each class is then reported, showing a nearly equal number of cases for each of the three configured classes.

```
(1000, 10) (1000,)
Counter({1: 334, 2: 334, 0: 332})
```

Listing 8.2: Example output from defining a synthetic multiclass classification dataset.

Next, we can demonstrate the built-in strategy of the `LogisticRegression` binary classification algorithm for handling more than two classes. The strategy for handling multiclass classification can be set via the `multi_class` argument and can be set to `''ovr''` for the One-vs-Rest strategy. The model will then be evaluated using repeated stratified k-fold cross-validation with three repeats and 10 folds, a good practice when evaluating machine learning models. We will summarize the performance of the model using the mean and standard deviation of classification accuracy across all repeats and folds. The complete example of evaluating a logistic regression model for multiclass classification using the built-in One-vs-Rest strategy is listed below.

```
# evaluate logistic regression for multi-class classification using built-in one-vs-rest
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.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    n_classes=3, random_state=1)
# define the model
model = LogisticRegression(multi_class='ovr')
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize the performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 8.3: Example of One-vs-Rest multiclass classification built-in to logistic regression.

Running the example defines the logistic regression model with the built-in handling of multiclass classification, then evaluates it on our synthetic multiclass classification dataset using the defined test procedure.

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

In this case, we can see that the model achieved a mean classification accuracy of about 68.6 percent.

```
Mean Accuracy: 0.686 (0.036)
```

Listing 8.4: Example output from evaluating logistic regression with the built-in OVR method on the synthetic multiclass classification dataset.

The scikit-learn library also provides a separate `OneVsRestClassifier` class that allows the One-vs-Rest strategy to be used with any classifier. This class can be used to harness a binary classifier like Logistic Regression or Perceptron for multiclass classification, or even other classifiers that natively support multiclass classification. It is very easy to use and requires that a classifier that is to be used for binary classification be provided to the `OneVsRestClassifier` as an argument. The example below shows how to evaluate the `OneVsRestClassifier` class with a `LogisticRegression` class used as the binary classification model.

```
# evaluate logistic regression for multi-class classification using one-vs-rest
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.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier
# define dataset
```

```
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    n_classes=3, random_state=1)
# define model
model = LogisticRegression()
# define the one-vs-rest strategy
ovr = OneVsRestClassifier(model)
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(ovr, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize the performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 8.5: Example of evaluating a standalone One-vs-Rest model for multiclass classification.

Running the example reports the performance of using the standalone One-vs-Rest strategy to wrap a logistic regression model.

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

In this case, we can see that the model achieved a mean classification accuracy of about 68.6 percent, matching the built-in usage of the strategy. This example can be used as the basis for using the One-vs-Rest strategy with any classification model.

```
Mean Accuracy: 0.686 (0.036)
```

Listing 8.6: Example output from evaluating logistic regression with the built-in OVR method on the synthetic multiclass classification dataset.

We can also use the `OneVsRestClassifier` wrapper class as a final model and make predictions for classification. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our multiclass classification dataset.

```
# make a prediction with logistic regression using one-vs-rest
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    n_classes=3, random_state=1)
# define model
model = LogisticRegression()
# define the one-vs-rest strategy
ovr = OneVsRestClassifier(model)
# fit the model on the whole dataset
ovr.fit(X, y)
# make a single prediction
row = [1.89149379, -0.39847585, 1.63856893, 0.01647165, 1.51892395, -3.52651223,
    1.80998823, 0.58810926, -0.02542177, -0.52835426]
yhat = ovr.predict([row])
print('Predicted Class: %d' % yhat[0])
```

Listing 8.7: Example of making a prediction with a One-vs-Rest wrapper with logistic regression for multiclass classification.

Running the example fits the One-vs-Rest algorithm on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 8.8: Example output from making a prediction with a One-vs-Rest wrapper with logistic regression for multiclass classification.

Now that we are familiar with using the One-vs-Rest strategy, let's look at the One-vs-One method.

8.4 One-Vs-One for Multiclass Classification

One-vs-One (OvO for short) is another heuristic method for using binary classification algorithms for multiclass classification. Like One-vs-Rest, One-vs-One splits a multiclass classification dataset into binary classification problems. Unlike One-vs-Rest that splits it into one binary dataset for each class, the One-vs-One approach splits the dataset into one dataset for each class versus every other class. For example, consider a multiclass classification problem with four classes: *red*, *blue*, and *green*, *yellow*. This could be divided into six binary classification datasets as follows:

- **Binary Classification Problem 1**: red vs. blue
- **Binary Classification Problem 2**: red vs. green
- **Binary Classification Problem 3**: red vs. yellow
- **Binary Classification Problem 4**: blue vs. green
- **Binary Classification Problem 5**: blue vs. yellow
- **Binary Classification Problem 6**: green vs. yellow

This is significantly more datasets, and in turn, more models than the One-vs-Rest strategy described in the previous section.

$$\text{Total Models} = \frac{\text{Total Classes} \times (\text{Total Classes} - 1)}{2} \tag{8.1}$$

We can see that for four classes, this gives us the expected value of six binary classification problems:

$$\begin{aligned} \text{Total Models} &= \frac{\text{Total Classes} \times (\text{Total Classes} - 1)}{2} \\ &= \frac{4 \times (4 - 1)}{2} \\ &= \frac{4 \times 3}{2} \\ &= \frac{12}{2} \\ &= 6 \end{aligned} \tag{8.2}$$

Each binary classification model may predict one class label and the class with the most predictions or votes is predicted by the One-vs-One strategy.

> An alternative is to introduce $K(K-1)/2$ binary discriminant functions, one for every possible pair of classes. This is known as a one-versus-one classifier. Each point is then classified according to a majority vote amongst the discriminant functions.
>
> — Page 183, *Pattern Recognition and Machine Learning*, 2006.

Similarly, if the binary classification models predict a numerical class membership, such as a probability, then the argmax of the sum of the scores (class with the largest sum score) is predicted as the class label. Classically, this approach is suggested for support vector machines (SVM) and related kernel-based algorithms. This is believed because the performance of kernel methods does not scale in proportion to the size of the training dataset and using subsets of the training data may counter this effect.

The support vector machine implementation in the scikit-learn is provided by the SVC class and supports the One-vs-One method for multiclass classification problems. This can be achieved by setting the `decision_function_shape` argument to `'ovo'`. The example below demonstrates how to evaluate the SVM algorithm for multiclass classification using the built-in One-vs-One method.

```
# evaluate SVM for multi-class classification using built-in one-vs-one
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.svm import SVC
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    n_classes=3, random_state=1)
# define the model
model = SVC(decision_function_shape='ovo')
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize the performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 8.9: Example of One-vs-One multiclass classification built-in to SVM.

Running the example defines the SVM model with the built-in handling of multiclass classification, then evaluates it on our synthetic multiclass classification dataset using the defined test procedure.

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

In this case, we can see that the model achieved a mean classification accuracy of about 87.1 percent.

```
Mean Accuracy: 0.871 (0.028)
```

Listing 8.10: Example output from evaluating SVM with the built-in OVO method on the synthetic multiclass classification dataset.

The scikit-learn library also provides a separate `OneVsOneClassifier` class that allows the One-vs-One strategy to be used with any classifier. This class can be used with a binary classifier like SVM, Logistic Regression or Perceptron for multiclass classification, or even other classifiers that natively support multiclass classification. It is very easy to use and requires that a classifier that is to be used for binary classification be provided to the `OneVsOneClassifier` as an argument. The example below demonstrates how to evaluate the `OneVsOneClassifier` class with an SVC class used as the binary classification model.

```
# evaluate SVM for multi-class classification using one-vs-one
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.svm import SVC
from sklearn.multiclass import OneVsOneClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    n_classes=3, random_state=1)
# define model
model = SVC()
# define the one-vs-one strategy
ovo = OneVsOneClassifier(model)
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(ovo, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize the performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 8.11: Example of standalone One-vs-One multiclass classification.

Running the example reports the performance of using the standalone One-vs-One strategy to wrap a SVM model.

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

In this case, we can see that the model achieved a mean classification accuracy of about 87.2 percent, very close to the built-in usage of the strategy for the SVM class. Again, this example can be used as the basis for using the One-vs-One strategy with any classification model.

```
Mean Accuracy: 0.872 (0.030)
```

Listing 8.12: Example output from evaluating SVM with the built-in OVO method on the synthetic multiclass classification dataset.

We can also use the `OneVsOneClassifier` wrapper class as a final model and make predictions for classification. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our multiclass classification dataset.

```
# make a prediction with SVM using one-vs-one
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.multiclass import OneVsOneClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5,
    n_classes=3, random_state=1)
# define model
model = SVC()
# define the one-vs-one strategy
ovo = OneVsOneClassifier(model)
# fit the model on the whole dataset
ovo.fit(X, y)
# make a single prediction
row = [1.89149379, -0.39847585, 1.63856893, 0.01647165, 1.51892395, -3.52651223,
    1.80998823, 0.58810926, -0.02542177, -0.52835426]
yhat = ovo.predict([row])
print('Predicted Class: %d' % yhat[0])
```

Listing 8.13: Example of making a prediction with a One-vs-One wrapper with SVM for multiclass classification.

Running the example fits the One-vs-One algorithm on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 8.14: Example output from making a prediction with a One-vs-One wrapper with SVM for multiclass classification.

8.5 Further Reading

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

Books

- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2RFWf3N
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/38wVdOd

APIs

- `sklearn.datasets.make_classification` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

- `sklearn.linear_model.LogisticRegression` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- `sklearn.svm.SVC` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
- Multiclass and multilabel algorithms, scikit-learn API.
  https://scikit-learn.org/stable/modules/multiclass.html
- `sklearn.multiclass.OneVsRestClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html
- `sklearn.multiclass.OneVsOneClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsOneClassifier.html

Articles

- Multiclass classification, Wikipedia.
  https://en.wikipedia.org/wiki/Multiclass_classification

8.6 Summary

In this tutorial, you discovered One-vs-Rest and One-vs-One strategies for multiclass classification. Specifically, you learned:

- Binary classification models like logistic regression and SVM do not support multiclass classification natively and require meta-strategies.
- The One-vs-Rest strategy splits a multiclass classification into one binary classification problem per class.
- The One-vs-One strategy splits a multiclass classification into one binary classification problem per each pair of classes.

Next

In the next section, we will take a closer look at the error correcting output codes multiple model algorithm for multi-class classification.

Chapter 9

Error-Correcting Output Codes

Some machine learning algorithms, like logistic regression and support vector machines, are designed for two-class (binary) classification problems. As such, these algorithms must either be modified for multiclass (more than two) classification problems or not used at all. The Error-Correcting Output Codes method is a technique that allows a multiclass classification problem to be reframed as multiple binary classification problems, allowing the use of native binary classification models to be used directly. Unlike One-vs-Rest and One-vs-One methods that offer a similar solution by dividing a multiclass classification problem into a fixed number of binary classification problems, the error-correcting output codes technique allows each class to be encoded as an arbitrary number of binary classification problems. When an overdetermined representation is used, it allows the extra models to act as *error-correction* predictions that can result in better predictive performance. In this tutorial, you will discover how to use error-correcting output codes for classification.

After completing this tutorial, you will know:

- Error-correcting output codes is a technique for using binary classification models on multiclass classification prediction tasks.
- How to fit, evaluate, and use error-correcting output codes classification models to make predictions.
- How to tune and evaluate different values for the number of bits per class hyperparameter used by error-correcting output codes.

Let's get started.

9.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Error-Correcting Output Codes
2. Evaluate ECOC Classifiers
3. Tune the Number of Bits Per Class

9.2 Error-Correcting Output Codes

Classification tasks are those where a label is predicted for a given input variable. Binary classification tasks are those classification problems where the target contains two values, whereas multiclass classification problems are those that have more than two target class labels. Many machine learning models have been developed for binary classification, although they may require modification to work with multiclass classification problems. For example, logistic regression and support vector machines were specifically designed for binary classification.

> Several machine learning algorithms, such as SVM, were originally designed to solve only binary classification tasks.
>
> — Page 133, *Pattern Classification Using Ensemble Methods*, 2010.

Rather than limiting the choice of algorithms or adapting the algorithms for multiclass problems, an alternative approach is to reframe the multiclass classification problem as multiple binary classification problems. Two common methods that can be used to achieve this include the One-vs-Rest (OvR) and One-vs-One (OvO) techniques (introduced in Chapter 8).

- **OvR**: splits a multiclass problem into one binary problem per class.
- **OvO**: splits a multiclass problem into one binary problem per each pair of classes.

Once split into subtasks, a binary classification model can be fit on each task and the model with the largest response can be taken as the prediction. Both the OvR and OvO may be thought of as a type of ensemble learning model given that multiple separate models are fit for a predictive modeling task and used in concert to make a prediction. In both cases, the prediction of the *ensemble members* is a simple winner take all approach.

> ... convert the multiclass task into an ensemble of binary classification tasks, whose results are then combined.
>
> — Page 134, *Pattern Classification Using Ensemble Methods*, 2010.

A related approach is to prepare a binary encoding (e.g. a bitstring) to represent each class in the problem. Each bit in the string can be predicted by a separate binary classification problem. Arbitrarily, length encodings can be chosen for a given multiclass classification problem. To be clear, each model receives the full input pattern and only predicts one position in the output string. During training, each model can be trained to produce the correct 0 or 1 output for the binary classification task. A prediction can then be made for new examples by using each model to make a prediction for the input to create the binary string, then compare the binary string to each class's known encoding. The class encoding that has the smallest distance to the prediction is then chosen as the output.

> A codeword of length l is ascribed to each class. Commonly, the size of the codewords has more bits than needed in order to uniquely represent each class.
>
> — Page 138, *Pattern Classification Using Ensemble Methods*, 2010.

It is an interesting approach that allows the class representation to be more elaborate than is required (perhaps overdetermined) as compared to a one hot encoding and introduces redundancy into the representation and modeling of the problem. This is intentional as the additional bits in the representation act like error-correcting codes to fix, correct, or improve the prediction.

> ... the idea is that the redundant "error-correcting" bits allow for some inaccuracies, and can improve performance.
>
> — Page 606, *The Elements of Statistical Learning*, 2016.

This gives the technique its name: error-correcting output codes, or ECOC for short.

> Error-Correcting Output Codes (ECOC) is a simple yet powerful approach to deal with a multi-class problem based on the combination of binary classifiers.
>
> — Page 90, *Ensemble Methods*, 2012.

Care can be taken to ensure that each encoded class has a very different binary string encoding. A suite of different encoding schemes has been explored as well as specific methods for constructing the encodings to ensure they are sufficiently far apart in the encoding space. Interestingly, random encodings have been found to work perhaps just as well.

> ... analyzed the ECOC approach, and showed that random code assignment worked as well as the optimally constructed error-correcting codes
>
> — Page 606, *The Elements of Statistical Learning*, 2016.

For a detailed review of the various different encoding schemes and methods for mapping predicted strings to encoded classes, I recommend Chapter 6 *Error Correcting Output Codes* of the book *Pattern Classification Using Ensemble Methods*.

9.3 Evaluate ECOC Classifiers

The scikit-learn library provides an implementation of ECOC via the `OutputCodeClassifier` class. The class takes as an argument the model to use to fit each binary classifier, and any machine learning model can be used. In this case, we will use a logistic regression model, intended for binary classification. The class also provides the `code_size` argument that specifies the size of the encoding for the classes as a multiple of the number of classes, e.g. the number of bits to encode for each class label. For example, if we wanted an encoding with bit strings with a length of 6 bits, and we had three classes, then we can specify the coding size as 2:

$$\begin{aligned} \text{Encoding Length} &= \text{Code Size} \times \text{Total Classes} \\ &= 2 \times 3 \\ &= 6 \end{aligned} \tag{9.1}$$

The example below demonstrates how to define an example of the `OutputCodeClassifier` with 2 bits per class and using a `LogisticRegression` model for each bit in the encoding.

```
...
# define the binary classification model
model = LogisticRegression()
# define the ecoc model
ecoc = OutputCodeClassifier(model, code_size=2, random_state=1)
```

Listing 9.1: Example of defining an ECOC model.

Although there are many sophisticated ways to construct the encoding for each class, the `OutputCodeClassifier` class selects a random bit string encoding for each class, at least at the time of writing. We can explore the use of the `OutputCodeClassifier` on a synthetic multiclass classification problem. We can use the `make_classification()` function to define a multiclass classification problem with 1,000 examples, 20 input features, and three classes. The example below demonstrates how to create the dataset and summarize the number of rows, columns, and classes in the dataset.

```
# multi-class classification dataset
from collections import Counter
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=1, n_classes=3)
# summarize the dataset
print(X.shape, y.shape)
# summarize the number of examples in each class
print(Counter(y))
```

Listing 9.2: Example of defining a synthetic multiclass classification dataset.

Running the example creates the dataset and reports the number of rows and columns, confirming the dataset was created as expected. The number of examples in each class is then reported, showing a nearly equal number of cases for each of the three configured classes.

```
(1000, 20) (1000,)
Counter({2: 335, 1: 333, 0: 332})
```

Listing 9.3: Example output from defining a synthetic multiclass classification dataset.

Next, we can evaluate an error-correcting output codes model on the dataset. We will use a logistic regression with 2 bits per class as we defined above. The model will then be evaluated using repeated stratified k-fold cross-validation with three repeats and 10 folds. We will summarize the performance of the model using the mean and standard deviation of classification accuracy across all repeats and folds.

```
...
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(ecoc, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize the performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 9.4: Example of defining the model evaluation procedure.

Tying this together, the complete example is listed below.

```
# evaluate error-correcting output codes for multi-class classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OutputCodeClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1, n_classes=3)
# define the binary classification model
model = LogisticRegression()
# define the ecoc model
ecoc = OutputCodeClassifier(model, code_size=2, random_state=1)
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(ecoc, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize the performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 9.5: Example of evaluating an ECOC model on the synthetic multiclass classification dataset.

Running the example defines the model and evaluates it on our synthetic multiclass classification dataset using the defined test procedure.

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

In this case, we can see that the model achieved a mean classification accuracy of about 76.6 percent.

```
Mean Accuracy: 0.766 (0.037)
```

Listing 9.6: Example output from evaluating an ECOC model on the synthetic multiclass classification dataset.

We may choose to use this as our final model. This requires that we fit the model on all available data and use it to make predictions on new data. The example below provides a full example of how to fit and use an error-correcting output model as a final model.

```
# use error-correcting output codes model as a final model and make a prediction
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OutputCodeClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1, n_classes=3)
# define the binary classification model
model = LogisticRegression()
# define the ecoc model
ecoc = OutputCodeClassifier(model, code_size=2, random_state=1)
```

```
# fit the model on the whole dataset
ecoc.fit(X, y)
# make a single prediction
row = [0.04339387, 2.75542632, -3.79522705, -0.71310994, -3.08888853, -1.2963487,
    -1.92065166, -3.15609907, 1.37532356, 3.61293237, 1.00353523, -3.77126962, 2.26638828,
    -10.22368666, -0.35137382, 1.84443763, 3.7040748, 2.50964286, 2.18839505, -2.31211692]
yhat = ecoc.predict([row])
print('Predicted Class: %d' % yhat[0])
```

Listing 9.7: Example of making a prediction with an ECOC model on the synthetic multiclass classification dataset.

Running the example fits the ECOC model on the entire dataset and uses the model to predict the class label for a single row of data. In this case, we can see that the model predicted the class label 0.

```
Predicted Class: 0
```

Listing 9.8: Example output from making a prediction with an ECOC model.

Now that we are familiar with how to fit and use the ECOC model, let's take a closer look at how to configure it.

9.4 Tune the Number of Bits Per Class

The key hyperparameter for the ECOC model is the encoding of class labels. This includes properties such as:

- The choice of representation (bits, real numbers, etc.)
- The encoding of each class label (random, etc.)
- The length of representation (number of bits, etc.)
- How predictions are mapped to classes (distance, etc.)

The `OutputCodeClassifier` scikit-learn implementation does not currently provide a lot of control over these elements. The element it does give control over is the number of bits used to encode each class label. In this section, we can perform a manual grid search across different numbers of bits per class label and compare the results. This provides a template that you can adapt and use on your own project. First, we can define a function to create and return the dataset.

```
# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=1, n_classes=3)
  return X, y
```

Listing 9.9: Example of a function for defining the dataset.

We can then define a function that will create a collection of models to evaluate. Each model will be an example of the `OutputCodeClassifier` using a `LogisticRegression` for each binary classification problem. We will configure the `code_size` of each model to be different, with values ranging from 1 to 20.

```
# get a list of models to evaluate
def get_models():
  models = dict()
  # enumerate the number of bits from 1 to 20
  for i in range(1,21):
    # create model
    model = LogisticRegression()
    # create error correcting output code classifier
    models[str(i)] = OutputCodeClassifier(model, code_size=i, random_state=1)
  return models
```

Listing 9.10: Example of a function for defining a suite of different configurations for the ECOC model.

We can evaluate each model using related k-fold cross-validation as we did in the previous section to give a sample of classification accuracy scores.

```
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the scores
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores
```

Listing 9.11: Example of a function for evaluating the performance of a model.

We can report the mean and standard deviation of the scores for each configuration and plot the distributions as box and whisker plots side by side to visually compare the results.

```
...
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the scores
  results.append(scores)
  names.append(name)
  # summarize results along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 9.12: Example of enumerating and evaluating ECOC model configurations.

Tying this all together, the complete example of comparing ECOC classification with a grid of the number of bits per class is listed below.

```
# compare the number of bits per class for error-correcting output code classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.multiclass import OutputCodeClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=1, n_classes=3)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # enumerate the number of bits from 1 to 20
  for i in range(1,21):
    # create model
    model = LogisticRegression()
    # create error correcting output code classifier
    models[str(i)] = OutputCodeClassifier(model, code_size=i, random_state=1)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the scores
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the scores
  results.append(scores)
  names.append(name)
  # summarize results along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 9.13: Example of evaluating ECOC models with differing numbers of bits per class.

Running the example first evaluates each model configuration and reports the mean and standard deviation of the accuracy scores.

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

In this case, we can see that perhaps 5 or 6 bits per class results in the best performance

with reported mean accuracy scores of about 78.2 percent and 78.0 percent respectively. We also see good results for 9, 13, 17, and 20 bits per class, with perhaps 17 bits per class giving the best result of about 78.5 percent.

```
>1 0.545 (0.032)
>2 0.766 (0.037)
>3 0.776 (0.036)
>4 0.769 (0.035)
>5 0.782 (0.037)
>6 0.780 (0.037)
>7 0.776 (0.039)
>8 0.775 (0.036)
>9 0.782 (0.038)
>10 0.779 (0.036)
>11 0.770 (0.033)
>12 0.777 (0.037)
>13 0.781 (0.037)
>14 0.779 (0.039)
>15 0.771 (0.033)
>16 0.769 (0.035)
>17 0.785 (0.034)
>18 0.776 (0.038)
>19 0.776 (0.034)
>20 0.780 (0.038)
```

Listing 9.14: Example output from evaluating ECOC models with differing numbers of bits per class.

A figure is created showing the box and whisker plots for the accuracy scores for each model configuration. We can see that besides a value of 1, the number of bits per class delivers similar results in terms of spread and mean accuracy scores that cluster around 77 percent. This suggests that the approach is reasonably stable across configurations.

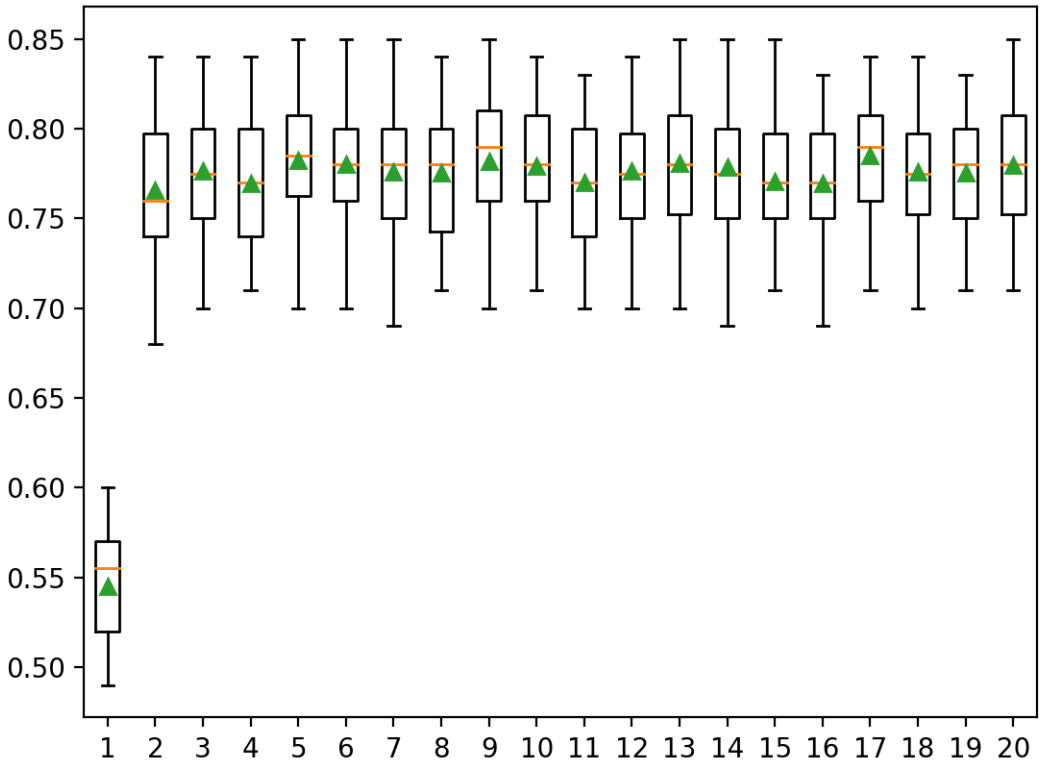

Figure 9.1: Box and Whisker Plots of Bits Per Class vs. Distribution of Classification Accuracy for ECOC.

9.5 Further Reading

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

Papers

- *Solving Multiclass Learning Problems via Error-Correcting Output Codes*, 1995.
  https://www.jair.org/index.php/jair/article/view/10127

Books

- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *The Elements of Statistical Learning*, 2016.
  https://amzn.to/31mzA31

APIs

- Error-Correcting Output-Codes, scikit-learn documentation.
  https://scikit-learn.org/stable/modules/multiclass.html#ecoc
- `sklearn.multiclass.OutputCodeClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OutputCodeClassifier.html

9.6 Summary

In this tutorial, you discovered how to use error-correcting output codes for classification. Specifically, you learned:

- Error-correcting output codes is a technique for using binary classification models on multiclass classification prediction tasks.
- How to fit, evaluate, and use error-correcting output codes classification models to make predictions.
- How to tune and evaluate different values for the number of bits per class hyperparameter used by error-correcting output codes.

Next

In the next section, we will take a closer look at multiple model machine learning algorithms for regression predictive modeling.

Chapter 10

Multiple Output Regression Models

Multiple-output regression, or multioutput regression for short are regression problems that involve predicting two or more numerical values given an input example. An example might be to predict a coordinate given an input, e.g. predicting x and y values. Another example would be multi-step time series forecasting that involves predicting multiple future time series of a given variable. Many machine learning algorithms are designed for predicting a single numeric value, referred to simply as regression. Some algorithms do support multioutput regression inherently, such as linear regression and decision trees. There are also special workaround models that can be used to wrap and use those algorithms that do not natively support predicting multiple outputs. These wrapper techniques may be considered a primitive type of ensemble model that combines multiple separate models. In this tutorial, you will discover how to develop machine learning models for multioutput regression. After completing this tutorial, you will know:

- The problem of multioutput regression in machine learning.
- How to develop machine learning models that inherently support multiple-output regression.
- How to develop wrapper models that allow algorithms that do not inherently support multiple outputs to be used for multiple-output regression.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. The Problem of Multioutput Regression
2. Inherently Multioutput Regression Algorithms
3. Wrapper Multioutput Regression Algorithms
4. Direct Multioutput Regression
5. Chained Multioutput Regression

10.2 The Problem of Multioutput Regression

Regression refers to a predictive modeling problem that involves predicting a numerical value. For example, predicting a size, weight, amount, number of sales, and number of clicks are regression problems. Typically, a single numeric value is predicted given input variables. Some regression problems require the prediction of two or more numeric values. For example, predicting an x and y coordinate. These problems are referred to as multiple-output regression, or multioutput regression.

- **Regression**: Predict a single numeric output given an input.
- **Multioutput Regression**: Predict two or more numeric outputs given an input.

In multioutput regression, typically the outputs are dependent upon the input and upon each other. This means that often the outputs are not independent of each other and may require a model that predicts both outputs together or each output contingent upon the other outputs. Multi-step time series forecasting may be considered a type of multiple-output regression where a sequence of future values are predicted and each predicted value is dependent upon the prior values in the sequence. There are a number of strategies for handling multioutput regression and we will explore some of them in this tutorial.

First, we can define a test problem that we can use to demonstrate the different modeling strategies. We will use the `make_regression()` function to create a test dataset for multiple-output regression. We will generate 1,000 examples with 10 input features, five of which will be redundant and five that will be informative. The problem will require the prediction of two numeric values.

- **Problem Input**: 10 numeric variables.
- **Problem Output**: 2 numeric variables.

The example below generates the dataset and summarizes the shape.

```
# example of multioutput regression dataset
from sklearn.datasets import make_regression
# create datasets
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# summarize dataset
print(X.shape, y.shape)
```

Listing 10.1: Example of creating the synthetic multioutput regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output elements of the dataset for modeling, confirming the chosen configuration.

```
(1000, 10) (1000, 2)
```

Listing 10.2: Example output from creating the synthetic multioutput regression dataset.

Next, let's look at modeling this problem directly.

10.3 Inherently Multioutput Regression Algorithms

Some regression machine learning algorithms support multiple outputs directly. This includes most of the popular machine learning algorithms implemented in the scikit-learn library, such as: `LinearRegression` (and related), `KNeighborsRegressor`, and the `DecisionTreeRegressor`. Let's look at a few examples to make this concrete.

10.3.1 Linear Regression for Multioutput Regression

The example below fits a linear regression model on the multioutput regression dataset, then makes a single prediction with the fit model.

```
# linear regression for multioutput regression
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
# create datasets
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define model
model = LinearRegression()
# fit model
model.fit(X, y)
# make a prediction
row = [0.21947749, 0.32948997, 0.81560036, 0.440956, -0.0606303, -0.29257894, -0.2820059,
    -0.00290545, 0.96402263, 0.04992249]
yhat = model.predict([row])
# summarize prediction
print(yhat[0])
```

Listing 10.3: Example of linear regression for multioutput regression.

Running the example fits the model and then makes a prediction for one input, confirming that the model predicted two required values.

```
[50.06781717 64.564973 ]
```

Listing 10.4: Example output from linear regression for multioutput regression.

10.3.2 k-Nearest Neighbors for Multioutput Regression

The example below fits a k-nearest neighbors model on the multioutput regression dataset, then makes a single prediction with the fit model.

```
# k-nearest neighbors for multioutput regression
from sklearn.datasets import make_regression
from sklearn.neighbors import KNeighborsRegressor
# create datasets
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define model
model = KNeighborsRegressor()
# fit model
model.fit(X, y)
# make a prediction
```

```
row = [0.21947749, 0.32948997, 0.81560036, 0.440956, -0.0606303, -0.29257894, -0.2820059,
    -0.00290545, 0.96402263, 0.04992249]
yhat = model.predict([row])
# summarize prediction
print(yhat[0])
```

Listing 10.5: Example of k-nearest neighbors for multioutput regression.

Running the example fits the model and then makes a prediction for one input, confirming that the model predicted two required values.

```
[-11.73511093 52.78406297]
```

Listing 10.6: Example output from k-nearest neighbors for multioutput regression.

10.3.3 Decision Tree for Multioutput Regression

The example below fits a decision tree model on the multioutput regression dataset, then makes a single prediction with the fit model.

```
# decision tree for multioutput regression
from sklearn.datasets import make_regression
from sklearn.tree import DecisionTreeRegressor
# create datasets
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define model
model = DecisionTreeRegressor()
# fit model
model.fit(X, y)
# make a prediction
row = [0.21947749, 0.32948997, 0.81560036, 0.440956, -0.0606303, -0.29257894, -0.2820059,
    -0.00290545, 0.96402263, 0.04992249]
yhat = model.predict([row])
# summarize prediction
print(yhat[0])
```

Listing 10.7: Example of decision tree for multioutput regression.

Running the example fits the model and then makes a prediction for one input, confirming that the model predicted two required values.

```
[49.93137149 64.08484989]
```

Listing 10.8: Example output from decision tree for multioutput regression.

10.3.4 Evaluate Multioutput Regression With Cross-Validation

We may want to evaluate a multioutput regression using k-fold cross-validation. This can be achieved in the same way as evaluating any other machine learning model. We will fit and evaluate a `DecisionTreeRegressor` model on the test problem using 10-fold cross-validation with three repeats. We will use the mean absolute error (MAE) performance metric as the score. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate multioutput regression model with k-fold cross-validation
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
# create datasets
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define model
model = DecisionTreeRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# summarize performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 10.9: Example of evaluating a decision tree for multioutput regression.

Running the example evaluates the performance of the decision tree model for multioutput regression on the test problem. The mean and standard deviation of the MAE is reported calculated across all folds and all repeats.

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

Importantly, error is reported across both output variables, rather than separate error scores for each output variable.

```
MAE: -52.410 (3.299)
```

Listing 10.10: Example output from evaluating a decision tree for multioutput regression.

10.4 Wrapper Multioutput Regression Algorithms

Not all regression algorithms support multioutput regression. One example is the support vector machine, although for regression, it is referred to as support vector regression, or SVR. This algorithm does not support multiple outputs for a regression problem and will raise an error. We can demonstrate this with an example, listed below.

```
# failure of support vector regression for multioutput regression (causes an error)
from sklearn.datasets import make_regression
from sklearn.svm import LinearSVR
# create datasets
```

```
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1)
# define model
model = LinearSVR()
# fit model
# (THIS WILL CAUSE AN ERROR!)
model.fit(X, y)
```

Listing 10.11: Example of the failure of SVM for multioutput regression.

Running the example reports an error message indicating that the model does not support multioutput regression.

```
ValueError: bad input shape (1000, 2)
```

Listing 10.12: Example output from the failure of SVM for multioutput regression.

A workaround for using regression models designed for predicting one value for multioutput regression is to divide the multioutput regression problem into multiple sub-problems. The most obvious way to do this is to split a multioutput regression problem into multiple single-output regression problems. For example, if a multioutput regression problem required the prediction of three values y_1, y_2 and y_3 given an input X, then this could be partitioned into three single-output regression problems:

- **Problem 1**: Given X, predict y_1.
- **Problem 2**: Given X, predict y_2.
- **Problem 3**: Given X, predict y_3.

There are two main approaches to implementing this technique. The first approach involves developing a separate regression model for each output value to be predicted. We can think of this as a direct approach, as each target value is modeled directly. The second approach is an extension of the first method except the models are organized into a chain. The prediction from the first model is taken as part of the input to the second model, and the process of output-to-input dependency repeats along the chain of models.

- **Direct Multioutput**: Develop an independent model for each numerical value to be predicted.
- **Chained Multioutput**: Develop a sequence of dependent models to match the number of numerical values to be predicted.

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

10.5 Direct Multioutput Regression

The direct approach to multioutput regression involves dividing the regression problem into a separate problem for each target variable to be predicted. This assumes that the outputs are independent of each other, which might not be a correct assumption. Nevertheless, this approach can provide surprisingly effective predictions on a range of problems and may be

worth trying, at least as a performance baseline. For example, the outputs for your problem may, in fact, be mostly independent, if not completely independent, and this strategy can help you find out. This approach is supported by the `MultiOutputRegressor` class that takes a regression model as an argument. It will then create one instance of the provided model for each output in the problem. The example below demonstrates how we can first create a single-output regression model then use the `MultiOutputRegressor` class to wrap the regression model and add support for multioutput regression.

```
...
# define base model
model = LinearSVR()
# define the direct multioutput wrapper model
wrapper = MultiOutputRegressor(model)
```

Listing 10.13: Example of using a direct multioutput regression model to wrap an SVM model.

We can demonstrate this strategy with a worked example on our synthetic multioutput regression problem. The example below demonstrates evaluating the `MultiOutputRegressor` class with linear SVR using repeated k-fold cross-validation and reporting the average mean absolute error (MAE) across all folds and repeats. The complete example is listed below.

```
# example of evaluating direct multioutput regression with an SVM model
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.multioutput import MultiOutputRegressor
from sklearn.svm import LinearSVR
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define base model
model = LinearSVR()
# define the direct multioutput wrapper model
wrapper = MultiOutputRegressor(model)
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(wrapper, X, y, scoring='neg_mean_absolute_error', cv=cv,
    n_jobs=-1)
# summarize performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 10.14: Example evaluating a direct multioutput regression wrapper model.

Running the example reports the mean and standard deviation MAE of the direct wrapper model.

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

In this case, we can see that the Linear SVR model wrapped by the direct multioutput regression strategy achieved a MAE of about 0.419.

```
MAE: -0.419 (0.024)
```

Listing 10.15: Example output from evaluating a direct multioutput regression wrapper model.

We can also use the direct multioutput regression wrapper as a final model and make predictions on new data. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our synthetic multioutput regression dataset.

```
# example of making a prediction with the direct multioutput regression model
from sklearn.datasets import make_regression
from sklearn.multioutput import MultiOutputRegressor
from sklearn.svm import LinearSVR
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define base model
model = LinearSVR()
# define the direct multioutput wrapper model
wrapper = MultiOutputRegressor(model)
# fit the model on the whole dataset
wrapper.fit(X, y)
# make a single prediction
row = [0.21947749, 0.32948997, 0.81560036, 0.440956, -0.0606303, -0.29257894, -0.2820059,
    -0.00290545, 0.96402263, 0.04992249]
yhat = wrapper.predict([row])
# summarize the prediction
print('Predicted: %s' % yhat[0])
```

Listing 10.16: Example of making a prediction using a direct multioutput regression wrapper model.

Running the example fits the direct wrapper model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted: [50.01932887 64.49432991]
```

Listing 10.17: Example output from making a prediction using a direct multioutput regression wrapper model.

Now that we are familiar with using the direct multioutput regression wrapper, let's look at the chained method.

10.6 Chained Multioutput Regression

Another approach to using single-output regression models for multioutput regression is to create a linear sequence of models. The first model in the sequence uses the input and predicts one output; the second model uses the input and the output from the first model to make a prediction; the third model uses the input and output from the first two models to make a prediction, and so on. For example, if a multioutput regression problem required the prediction of three values y_1, y_2 and y_3 given an input X, then this could be partitioned into three dependent single-output regression problems:

- **Problem 1**: Given X, predict y_1.
- **Problem 2**: Given X and $yhat_1$, predict y_2.
- **Problem 3**: Given X, $yhat_1$, and $yhat_2$, predict y_3.

This can be achieved using the `RegressorChain` class in the scikit-learn library. The order of the models may be based on the order of the outputs in the dataset (the default) or specified via the `order` argument. For example, `order=[0,1]` would first predict the 0^{th} output, then the 1^{st} output, whereas an `order=[1,0]` would first predict the last output variable and then the first output variable in our test problem. The example below demonstrates how we can first create a single-output regression model then use the `RegressorChain` class to wrap the regression model and add support for multioutput regression.

```
...
# define base model
model = LinearSVR()
# define the chained multioutput wrapper model
wrapper = RegressorChain(model, order=[0,1])
```

Listing 10.18: Example of using a chained multioutput regression model to wrap an SVM model.

We can demonstrate this strategy with a worked example on our synthetic multioutput regression problem. The example below demonstrates evaluating the `RegressorChain` class with linear SVR using repeated k-fold cross-validation and reporting the average mean absolute error (MAE) across all folds and repeats. The complete example is listed below.

```
# example of evaluating chained multioutput regression with an SVM model
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.multioutput import RegressorChain
from sklearn.svm import LinearSVR
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define base model
model = LinearSVR()
# define the chained multioutput wrapper model
wrapper = RegressorChain(model)
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the scores
n_scores = cross_val_score(wrapper, X, y, scoring='neg_mean_absolute_error', cv=cv,
    n_jobs=-1)
# summarize performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 10.19: Example evaluating a chained multioutput regression wrapper model.

Running the example reports the mean and standard deviation MAE of the chained wrapper model. Note that you may see a `ConvergenceWarning` when running the example, which can be safely ignored.

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

In this case, we can see that the Linear SVR model wrapped by the chained multioutput regression strategy achieved a MAE of about 0.643.

```
MAE: -0.643 (0.313)
```

Listing 10.20: Example output from evaluating a chained multioutput regression wrapper model.

We can also use the chained multioutput regression wrapper as a final model and make predictions on new data. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our synthetic multioutput regression dataset.

```
# example of making a prediction with the chained multioutput regression model
from sklearn.datasets import make_regression
from sklearn.multioutput import RegressorChain
from sklearn.svm import LinearSVR
# define dataset
X, y = make_regression(n_samples=1000, n_features=10, n_informative=5, n_targets=2,
    random_state=1, noise=0.5)
# define base model
model = LinearSVR()
# define the chained multioutput wrapper model
wrapper = RegressorChain(model)
# fit the model on the whole dataset
wrapper.fit(X, y)
# make a single prediction
row = [0.21947749, 0.32948997, 0.81560036, 0.440956, -0.0606303, -0.29257894, -0.2820059,
    -0.00290545, 0.96402263, 0.04992249]
yhat = wrapper.predict([row])
# summarize the prediction
print('Predicted: %s' % yhat[0])
```

Listing 10.21: Example of making a prediction using a chained multioutput regression wrapper model.

Running the example fits the chained wrapper model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted: [50.03206  64.73673318]
```

Listing 10.22: Example output from making a prediction using a chained multioutput regression wrapper model.

10.7 Further Reading

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

APIs

- Multiclass and multilabel algorithms, API.
  https://scikit-learn.org/stable/modules/multiclass.html
- `sklearn.datasets.make_regression` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html
- `sklearn.multioutput.MultiOutputRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputRegressor.html
- `sklearn.multioutput.RegressorChain` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.RegressorChain.html

10.8 Summary

In this tutorial, you discovered how to develop machine learning models for multioutput regression. Specifically, you learned:

- The problem of multioutput regression in machine learning.
- How to develop machine learning models that inherently support multiple-output regression.
- How to develop wrapper models that allow algorithms that do not inherently support multiple outputs to be used for multiple-output regression.

Next

In the next section, we will take a closer look at dynamic classifier selection that uses multiple classifier models.

Chapter 11

Dynamic Classifier Selection

Dynamic Classifier Selection is a type of ensemble learning algorithm for classification predictive modeling. The technique involves fitting multiple machine learning models on the training dataset, then selecting the model that is expected to perform best when making a prediction, based on the specific details of the example to be predicted. This can be achieved using a k-nearest neighbor model to locate examples in the training dataset that are closest to the new example to be predicted, evaluating all models in the pool on this neighborhood and using the model that performs the best on the neighborhood to make a prediction for the new example.

As such, the dynamic classifier selection can often perform better than any single model in the pool and provides an alternative to averaging the predictions from multiple models, as is the case in other ensemble algorithms. In this tutorial, you will discover how to develop dynamic classifier selection ensembles in Python. After completing this tutorial, you will know:

- Dynamic classifier selection algorithms choose one from among many models to make a prediction for each new example.
- How to develop and evaluate dynamic classifier selection models for classification tasks using the scikit-learn API.
- How to explore the effect of dynamic classifier selection model hyperparameters on classification accuracy.

Let's get started.

11.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Dynamic Classifier Selection
2. Evaluate Dynamic Classifier Selection Models
3. Hyperparameter Tuning for DCS

11.2 Dynamic Classifier Selection

Multiple Classifier Systems refers to a field of machine learning algorithms that use multiple models to address classification predictive modeling problems. This includes familiar techniques such as One-vs-Rest, One-vs-One, and output error-correcting codes techniques. It also includes more general techniques that select a model to use dynamically for each new example that requires a prediction.

> Several approaches are currently used to construct an MCS [...] One of the most promising MCS approaches is Dynamic Selection (DS), in which the base classifiers are selected on the fly, according to each new sample to be classified.
>
> — *Dynamic Classifier Selection: Recent Advances And Perspectives*, 2018.

These methods are generally known by the name: Dynamic Classifier Selection, or DCS for short.

- **Dynamic Classifier Selection**: Algorithms that choose one from among many trained models to make a prediction based on the specific details of the input.

Given that multiple models are used in DCS, it is considered a type of ensemble learning technique. Dynamic Classifier Selection algorithms generally involve partitioning the input feature space in some way and assigning specific models to be responsible for making predictions for each partition. There are a variety of different DCS algorithms and research efforts are mainly focused on how to evaluate and assign classifiers to specific regions of the input space.

> After training multiple individual learners, DCS dynamically selects one learner for each test instance. [...] DCS makes predictions by using one individual learner.
>
> — Page 93, *Ensemble Methods: Foundations and Algorithms*, 2012.

An early and popular approach involves first fitting a small, diverse set of classification models on the training dataset. When a prediction is required, first a k-nearest neighbor (kNN) algorithm is used to find the k most similar examples from the training dataset that match the example. Each previously fit classifier in the model is then evaluated on the neighbor of k training examples and the classifier that performs the best is selected to make a prediction for the new example. This approach is referred to as *Dynamic Classifier Selection Local Accuracy* or DCS-LA for short and was described by Kevin Woods, et al. in their 1997 paper titled *Combination Of Multiple Classifiers Using Local Accuracy Estimates*.

> The basic idea is to estimate each classifier's accuracy in local region of feature space surrounding an unknown test sample, and then use the decision of the most locally accurate classifier.
>
> — *Combination Of Multiple Classifiers Using Local Accuracy Estimates*, 1997.

The authors describe two approaches for selecting a single classifier model to make a prediction for a given input example, they are:

- **Local Accuracy**, often referred to as LA or Overall Local Accuracy (OLA).
- **Class Accuracy**, often referred to as CA or Local Class Accuracy (LCA).

Local Accuracy (OLA) involves evaluating the classification accuracy of each model on the neighborhood of k training examples. The model that performs the best in this neighborhood is then selected to make a prediction for the new example.

> The OLA of each classifier is computed as the percentage of the correct recognition of the samples in the local region.
>
> — *Dynamic Selection Of Classifiers: A Comprehensive Review*, 2014.

Class Accuracy (LCA) involves using each model to make a prediction for the new example and noting the class that was predicted. Then, the accuracy of each model on the neighbor of k training examples is evaluated and the model that has the best skill for the class that it predicted on the new example is selected and its prediction returned.

> The LCA is estimated for each base classifier as the percentage of correct classifications within the local region, but considering only those examples where the classifier has given the same class as the one it gives for the unknown pattern.
>
> — *Dynamic Selection Of Classifiers: A Comprehensive Review*, 2014.

In both cases, if all fit models make the same prediction for a new input example, then the prediction is returned directly. Now that we are familiar with DCS and the DCS-LA algorithm, let's look at how we can use it on our own classification predictive modeling projects.

11.3 Evaluate Dynamic Classifier Selection Models

The Dynamic Ensemble Selection Library or DESlib for short is an open source Python library that provides an implementation of many different dynamic classifier selection algorithms. DESlib is an easy-to-use ensemble learning library focused on the implementation of the state-of-the-art techniques for dynamic classifier and ensemble selection. First, we can install the DESlib library using the `pip` package manager.

```
pip install deslib
```

Listing 11.1: Example of installing the DESlib library with `pip`.

Once installed, we can confirm that the library was installed correctly and is ready to be used by loading the library and printing the installed version.

```
# check deslib version
import deslib
print(deslib.__version__)
```

Listing 11.2: Example of printing the version of the DESlib library.

Running the script will print your version of the DESlib library you have installed. Your version should be the same or higher. If not, you must upgrade your version of the DESlib library.

```
0.3
```

Listing 11.3: Example output from printing the version of the DESlib library.

The DESlib provides an implementation of the DCS-LA algorithm with each classifier selection technique via the `OLA` and `LCA` classes respectively. Each class can be used as a scikit-learn model directly, allowing the full suite of scikit-learn data preparation, modeling pipelines, and model evaluation techniques to be used. Both classes use a k-nearest neighbor algorithm to select the neighbor with a default value of $k = 7$. A bootstrap aggregation (bagging) ensemble of decision trees is used as the pool of classifier models considered for each classification that is made by default, although this can be changed by setting `pool_classifiers` to a list of models. We can use the `make_classification()` function to create a synthetic binary classification problem with 10,000 examples and 20 input features.

```
# synthetic binary classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 11.4: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(10000, 20) (10000,)
```

Listing 11.5: Example output from creating the synthetic classification dataset.

Now that we are familiar with the DESlib API, let's look at how to use each DCS-LA algorithm.

11.3.1 DCS With Overall Local Accuracy (OLA)

We can evaluate a DCS-LA model using overall local accuracy on the synthetic dataset. In this case, we will use default model hyperparameters, including bagged decision trees as the pool of classifier models and a $k = 7$ for the selection of the local neighborhood when making a prediction. We will evaluate the model using repeated stratified k-fold cross-validation with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds. The complete example is listed below.

```
# evaluate dynamic classifier selection DCS-LA with overall local accuracy
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 deslib.dcs.ola import OLA
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
```

```
model = OLA()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 11.6: Example of evaluating DCS with OLA on the classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the DCS-LA with OLA and default hyperparameters achieves a classification accuracy of about 88.3 percent.

```
Mean Accuracy: 0.883 (0.012)
```

Listing 11.7: Example output from evaluating DCS with OLA on the classification dataset.

We can also use the DCS-LA model with OLA as a final model and make predictions for classification. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with DCS-LA using overall local accuracy
from sklearn.datasets import make_classification
from deslib.dcs.ola import OLA
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = OLA()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719, 0.28422388,
    -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799, 3.34692332,
    4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict([row])
# summarize the prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 11.8: Example of making a prediction using a DCS with OLA model.

Running the example fits the DCS-LA with OLA model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 0
```

Listing 11.9: Example output from making a prediction using a DCS with OLA model.

Now that we are familiar with using DCS-LA with OLA, let's look at the LCA method.

11.3.2 DCS With Local Class Accuracy (LCA)

We can evaluate a DCS-LA model using local class accuracy on the synthetic dataset. In this case, we will use default model hyperparameters, including bagged decision trees as the pool of classifier models and a $k = 7$ for the selection of the local neighborhood when making a prediction. We will evaluate the model using repeated stratified k-fold cross-validation with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds. The complete example is listed below.

```
# evaluate dynamic classifier selection DCS-LA using local class accuracy
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 deslib.dcs.lca import LCA
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = LCA()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 11.10: Example of evaluating DCS with LCA on the classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the DCS-LA with LCA and default hyperparameters achieves a classification accuracy of about 92.2 percent.

```
Mean Accuracy: 0.922 (0.007)
```

Listing 11.11: Example output from evaluating DCS with LCA on the classification dataset.

We can also use the DCS-LA model with LCA as a final model and make predictions for classification. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with DCS-LA using local class accuracy
from sklearn.datasets import make_classification
from deslib.dcs.lca import LCA
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
```

```
model = LCA()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719, 0.28422388,
    -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799, 3.34692332,
    4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict([row])
# summarize the prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 11.12: Example of making a prediction using a DCS with LCA model.

Running the example fits the DCS-LA with LCA model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 0
```

Listing 11.13: Example output from making a prediction using a DCS with LCA model.

Now that we are familiar with using the scikit-learn API to evaluate and use DCS-LA models, let's look at configuring the model.

11.4 Hyperparameter Tuning for DCS

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the DCS-LA model and their effect on model performance. There are many hyperparameters we can look at for DCS-LA, although in this case, we will look at the value of k in the k-nearest neighbor model used in the local evaluation of the models, and how to use a custom pool of classifiers. We will use the DCS-LA with OLA as the basis for these experiments, although the choice of the specific method is arbitrary.

11.4.1 Explore k in k-Nearest Neighbors

The configuration of the k-nearest neighbors algorithm is critical to the DCS-LA model as it defines the scope of the neighborhood in which each classifier is considered for selection. The k value controls the size of the neighborhood and it is important to set it to a value that is appropriate for your dataset, specifically the density of samples in the feature space. A value too small will mean that relevant examples in the training set might be excluded from the neighborhood, whereas values too large may mean that the signal is being washed out by too many examples. The example below explores the classification accuracy of the DCS-LA with OLA with k values from 2 to 21.

```
# explore k in knn for DCS-LA with overall local accuracy
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 deslib.dcs.ola import OLA
from matplotlib import pyplot
```

```
# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=10000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # evaluate k values from 2 to 21
  for n in range(2,22):
    models[str(n)] = OLA(k=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the scores
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize results along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 11.14: Example of evaluating k values for the DCS model.

Running the example first reports the mean accuracy for each configured neighborhood size.

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

In this case, we can see that accuracy increases with the neighborhood size, perhaps to $k = 13$ or $k = 14$, where it appears to level off.

```
>2 0.873 (0.009)
>3 0.874 (0.013)
>4 0.880 (0.009)
>5 0.881 (0.009)
```

```
>6 0.883 (0.010)
>7 0.883 (0.011)
>8 0.884 (0.012)
>9 0.883 (0.010)
>10 0.886 (0.012)
>11 0.886 (0.011)
>12 0.885 (0.010)
>13 0.888 (0.010)
>14 0.886 (0.009)
>15 0.889 (0.010)
>16 0.885 (0.012)
>17 0.888 (0.009)
>18 0.886 (0.010)
>19 0.889 (0.012)
>20 0.889 (0.011)
>21 0.886 (0.011)
```

Listing 11.15: Example output from evaluating k values for the DCS model.

A box and whisker plot is created for the distribution of accuracy scores for each configured neighborhood size. We can see the general trend of increasing model performance and k value before reaching a plateau.

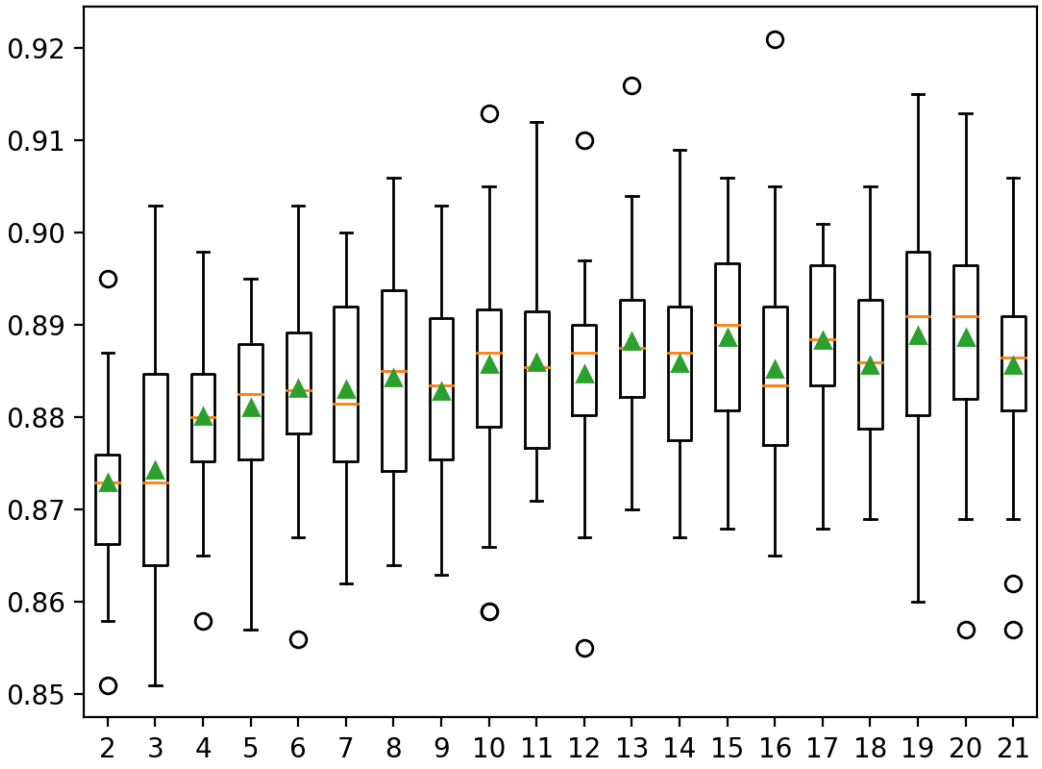

Figure 11.1: Box and Whisker Plots of Accuracy Distributions for k Values in DCS-LA With OLA.

11.4.2 Explore Algorithms for Classifier Pool

The choice of algorithms used in the pool for the DCS-LA is another important hyperparameter. By default, bagged decision trees are used, as it has proven to be an effective approach on a range of classification tasks. Nevertheless, a custom pool of classifiers can be considered. This requires first defining a list of classifier models to use and fitting each on the training dataset. Unfortunately, this means that the automatic k-fold cross-validation model evaluation methods in scikit-learn cannot be used in this case. Instead, we will use a train-test split so that we can fit the classifier pool manually on the training dataset.

The list of fit classifiers can then be specified to the OLA (or LCA) class via the `pool_classifiers` argument. In this case, we will use a pool that includes logistic regression, a decision tree, and a naive Bayes classifier. The complete example of evaluating DCS-LA with OLA and a custom set of classifiers on the synthetic dataset is listed below.

```
# evaluate DCS-LA using OLA with a custom pool of algorithms
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from deslib.dcs.ola import OLA
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
# define classifiers to use in the pool
classifiers = [
  LogisticRegression(),
  DecisionTreeClassifier(),
  GaussianNB()]
# fit each classifier on the training set
for c in classifiers:
  c.fit(X_train, y_train)
# define the DCS-LA model
model = OLA(pool_classifiers=classifiers)
# fit the model
model.fit(X_train, y_train)
# make predictions on the test set
yhat = model.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Accuracy: %.3f' % (score))
```

Listing 11.16: Example of evaluating custom pool classifiers for the DCS model.

Running the example first reports the mean accuracy for the model with the custom pool of classifiers.

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

In this case, we can see that the model achieved an accuracy of about 91.2 percent.

```
Accuracy: 0.913
```

Listing 11.17: Example output from evaluating custom pool classifiers for the DCS model.

In order to adopt the DCS model, it must perform better than any contributing model. Otherwise, we would simply use the contributing model that performs better instead. We can check this by evaluating the performance of each contributing classifier on the test set.

```
...
# evaluate contributing models
for c in classifiers:
  yhat = c.predict(X_test)
  score = accuracy_score(y_test, yhat)
  print('>%s: %.3f' % (c.__class__.__name__, score))
```

Listing 11.18: Example of evaluating standalone classifiers used in the DCS pool.

The updated example of DCS-LA with a custom pool of classifiers that are also evaluated independently is listed below.

```
# evaluate DCS-LA using OLA with a custom pool of algorithms
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from deslib.dcs.ola import OLA
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
# define classifiers to use in the pool
classifiers = [
  LogisticRegression(),
  DecisionTreeClassifier(),
  GaussianNB()]
# fit each classifier on the training set
for c in classifiers:
  c.fit(X_train, y_train)
# define the DCS-LA model
model = OLA(pool_classifiers=classifiers)
# fit the model
model.fit(X_train, y_train)
# make predictions on the test set
yhat = model.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Accuracy: %.3f' % (score))
# evaluate contributing models
for c in classifiers:
  yhat = c.predict(X_test)
  score = accuracy_score(y_test, yhat)
  print('>%s: %.3f' % (c.__class__.__name__, score))
```

Listing 11.19: Example of evaluating standalone classifiers used in the DCS pool.

Running the example first reports the mean accuracy for the model with the custom pool of classifiers and the accuracy of each contributing model.

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

In this case, we can see that again, the DCS-LA achieves an accuracy of about 91.3 percent, which is better than any contributing model.

```
Accuracy: 0.913
>LogisticRegression: 0.878
>DecisionTreeClassifier: 0.884
>GaussianNB: 0.873
```

Listing 11.20: Example output from evaluating standalone classifiers used in the DCS pool.

11.5 Further Reading

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

Papers

- *Combination Of Multiple Classifiers Using Local Accuracy Estimates*, 1997.
  https://ieeexplore.ieee.org/abstract/document/588027
- *Dynamic Selection Of Classifiers: A Comprehensive Review*, 2014.
  https://www.sciencedirect.com/science/article/abs/pii/S0031320314001885
- *Dynamic Classifier Selection: Recent Advances And Perspectives*, 2018.
  https://www.sciencedirect.com/science/article/pii/S1566253517304074

Books

- *Ensemble Methods: Foundations and Algorithms*, 2012.
  https://amzn.to/32L1yWD

APIs

- Dynamic Selection Library Project, GitHub.
  https://github.com/scikit-learn-contrib/DESlib
- DESlib API Documentation.
  https://deslib.readthedocs.io/en/latest/api.html

11.6 Summary

In this tutorial, you discovered how to develop dynamic classifier selection ensembles in Python. Specifically, you learned:

- Dynamic classifier selection algorithms choose one from among many models to make a prediction for each new example.
- How to develop and evaluate dynamic classifier selection models for classification tasks using the scikit-learn API.
- How to explore the effect of dynamic classifier selection model hyperparameters on classification accuracy.

Next

In the next section, we will take a closer look at dynamic ensemble selection that is a primitive type of ensemble machine learning.

Chapter 12

Dynamic Ensemble Selection

Dynamic ensemble selection is an ensemble learning technique that automatically selects a subset of ensemble members just-in-time when making a prediction. The technique involves fitting multiple machine learning models on the training dataset, then selecting the models that are expected to perform best when making a prediction for a specific new example, based on the details of the example to be predicted.

This can be achieved using a k-nearest neighbor model to locate examples in the training dataset that are closest to the new example to be predicted, evaluating all models in the pool on this neighborhood and using the models that perform the best on the neighborhood to make a prediction for the new example. As such, the dynamic ensemble selection can often perform better than any single model in the pool and better than averaging all members of the pool, so-called static ensemble selection. In this tutorial, you will discover how to develop dynamic ensemble selection models in Python. After completing this tutorial, you will know:

- Dynamic ensemble selection algorithms automatically choose ensemble members when making a prediction on new data.
- How to develop and evaluate dynamic ensemble selection models for classification tasks using the scikit-learn API.
- How to explore the effect of dynamic ensemble selection model hyperparameters on classification accuracy.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Dynamic Ensemble Selection
2. Evaluate KNORA Models
3. Hyperparameter Tuning for KNORA

12.2 Dynamic Ensemble Selection

Multiple Classifier Systems refer to a field of machine learning algorithms that use multiple models to address classification predictive modeling problems. The first class of multiple classifier systems to find success is referred to as Dynamic Classifier Selection, or DCS for short (introduced in Chapter 11).

- **Dynamic Classifier Selection**: Algorithms that dynamically choose one from among many trained models to make a prediction based on the specific details of the input.

Dynamic Classifier Selection algorithms generally involve partitioning the input feature space in some way and assigning specific models to be responsible for making predictions for each partition. There are a variety of different DCS algorithms and research efforts are mainly focused on how to evaluate and assign classifiers to specific regions of the input space.

> After training multiple individual learners, DCS dynamically selects one learner for each test instance. [...] DCS makes predictions by using one individual learner.
>
> — Page 93, *Ensemble Methods: Foundations and Algorithms*, 2012.

A natural extension to DCS is algorithms that select one or more models dynamically in order to make a prediction. That is, selecting a subset or ensemble of classifiers dynamically. These techniques are referred to as dynamic ensemble selection, or DES.

- **Dynamic Ensemble Selection**: Algorithms that dynamically choose a subset of trained models to make a prediction based on the specific details of the input.

Dynamic Ensemble Selection algorithms operate much like DCS algorithms, except predictions are made using votes from multiple classifier models instead of a single best model. In effect, each region of the input feature space is owned by a subset of models that perform best in that region.

> ... given the fact that selecting only one classifier can be highly error-prone, some researchers decided to select a subset of the pool of classifiers rather than just a single base classifier. All base classifiers that obtained a certain competence level are used to compose the EoC, and their outputs are aggregated to predict the label ...
>
> — *Dynamic Classifier Selection: Recent Advances And Perspectives*, 2018.

Perhaps the canonical approach to dynamic ensemble selection is the k-Nearest Neighbor Oracle, or KNORA, algorithm as it is a natural extension of the canonical dynamic classifier selection algorithm *Dynamic Classifier Selection Local Accuracy*, or DCS-LA. DCS-LA involves selecting the k-nearest neighbors from the training or validation dataset for a given new input pattern, then selecting the single best classifier based on its performance in that neighborhood of k examples to make a prediction on the new example. KNORA was described by Albert Ko, et al. in their 2008 paper titled *From Dynamic Classifier Selection To Dynamic Ensemble Selection*. It is an extension of DCS-LA that selects multiple models that perform well on the neighborhood and whose predictions are then combined using majority voting to make a final output prediction.

> For any test data point, KNORA simply finds its nearest K neighbors in the validation set, figures out which classifiers correctly classify those neighbors in the validation set and uses them as the ensemble for classifying the given pattern in that test set.
>
> — *From Dynamic Classifier Selection To Dynamic Ensemble Selection*, 2008.

The selected classifier models are referred to as *oracles*, hence the use of oracle in the name of the method. The ensemble is considered dynamic because the members are chosen just-in-time conditional on the specific input pattern requiring a prediction. This is opposed to static, where ensemble members are chosen once, such as averaging predictions from all classifiers in the model.

> This is done through a dynamic fashion, since different patterns might require different ensembles of classifiers. Thus, we call our method a dynamic ensemble selection.
>
> — *From Dynamic Classifier Selection To Dynamic Ensemble Selection*, 2008.

Two versions of KNORA are described, including KNORA-Eliminate and KNORA-Union.

- **KNORA-Eliminate (KNORA-E)**: Ensemble of classifiers that achieves perfect accuracy on the neighborhood of the new example, with a reducing neighborhood size until at least one perfect classifier is located.
- **KNORA-Union (KNORA-U)**: Ensemble of all classifiers that makes at least one correct prediction on the neighborhood with weighted voting and votes proportional to accuracy on the neighborhood.

KNORA-Eliminate, or KNORA-E for short, involves selecting all classifiers that achieve perfect predictions on the neighborhood of k examples in the neighborhood. If no classifier achieves 100 percent accuracy, the neighborhood size is reduced by one and the models are re-evaluated. This process is repeated until one or more models are discovered that has perfect performance, and then used to make a prediction for the new example.

> In the case where no classifier can correctly classify all the K-nearest neighbors of the test pattern, then we simply decrease the value of K until at least one classifier correctly classifies its neighbors.
>
> — *From Dynamic Classifier Selection To Dynamic Ensemble Selection*, 2008.

KNORA-Union, or KNORA-U for short, involves selecting all classifiers that make at least one correct prediction in the neighborhood. The predictions from each classifier are then combined using a weighted average, where the number of correct predictions in the neighborhood indicates the number of votes assigned to each classifier.

> The more neighbors a classifier classifies correctly, the more votes this classifier will have for a test pattern.
>
> — *From Dynamic Classifier Selection To Dynamic Ensemble Selection*, 2008.

Now that we are familiar with DES and the KNORA algorithm, let's look at how we can use it on our own classification predictive modeling projects.

12.3 Evaluate KNORA Models

The Dynamic Ensemble Library, or DESlib for short, is a Python machine learning library that provides an implementation of many different dynamic classifier and dynamic ensemble selection algorithms. DESlib is an easy-to-use ensemble learning library focused on the implementation of the state-of-the-art techniques for dynamic classifier and ensemble selection. For installation instructions for the DESlib, see Section 11.3. The DESlib provides an implementation of the KNORA algorithm with each dynamic ensemble selection technique via the `KNORAE` and `KNORAU` classes respectively.

Each class can be used as a scikit-learn model directly, allowing the full suite of scikit-learn data preparation, modeling pipelines, and model evaluation techniques to be used directly. Both classes use a k-nearest neighbor algorithm to select the neighbor with a default value of $k = 7$. A bootstrap aggregation (bagging) ensemble of decision trees is used as the pool of classifier models considered for each classification that is made by default, although this can be changed by setting `pool_classifiers` to a list of models. We can use the `make_classification()` function to create a synthetic binary classification problem with 10,000 examples and 20 input features.

```
# synthetic binary classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 12.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(10000, 20) (10000,)
```

Listing 12.2: Example output from creating the synthetic classification dataset.

Now that we are familiar with the DESlib API, let's look at how to use each KNORA algorithm on our synthetic classification dataset.

12.3.1 KNORA-Eliminate (KNORA-E)

We can evaluate a KNORA-Eliminate dynamic ensemble selection algorithm on the synthetic dataset. In this case, we will use default model hyperparameters, including bagged decision trees as the pool of classifier models and $k = 7$ for the selection of the local neighborhood when making a prediction. We will evaluate the model using repeated stratified k-fold cross-validation with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds. The complete example is listed below.

```
# evaluate dynamic KNORA-E dynamic ensemble selection for binary classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
```

```
from deslib.des.knora_e import KNORAE
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = KNORAE()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 12.3: Example of evaluating KNORA-E on the classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the KNORA-E ensemble and default hyperparameters achieve a classification accuracy of about 91.5 percent.

```
Mean Accuracy: 0.915 (0.009)
```

Listing 12.4: Example output from evaluating KNORA-E on the classification dataset.

We can also use the KNORA-E ensemble as a final model and make predictions for classification. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with KNORA-E dynamic ensemble selection
from sklearn.datasets import make_classification
from deslib.des.knora_e import KNORAE
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = KNORAE()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719, 0.28422388,
    -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799, 3.34692332,
    4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict([row])
# summarize the prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 12.5: Example of making a prediction with a KNORA-E model.

Running the example fits the KNORA-E dynamic ensemble selection algorithm on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 12.6: Example output from making a prediction with a KNORA-E model.

Now that we are familiar with using KNORA-E, let's look at the KNORA-Union method.

12.3.2 KNORA-Union (KNORA-U)

We can evaluate a KNORA-Union model on the synthetic dataset. In this case, we will use default model hyperparameters, including bagged decision trees as the pool of classifier models and a $k = 7$ for the selection of the local neighborhood when making a prediction. We will evaluate the model using repeated stratified k-fold cross-validation with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds. The complete example is listed below.

```
# evaluate dynamic KNORA-U dynamic ensemble selection for binary classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from deslib.des.knora_u import KNORAU
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = KNORAU()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 12.7: Example of evaluating KNORA-U on the classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the KNORA-U dynamic ensemble selection model and default hyperparameters achieve a classification accuracy of about 93.3 percent.

```
Mean Accuracy: 0.933 (0.009)
```

Listing 12.8: Example output from evaluating KNORA-U on the classification dataset.

We can also use the KNORA-U model as a final model and make predictions for classification. First, the model is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with KNORA-U dynamic ensemble selection
from sklearn.datasets import make_classification
from deslib.des.knora_u import KNORAU
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = KNORAU()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719, 0.28422388,
    -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799, 3.34692332,
    4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict([row])
# summarize the prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 12.9: Example of making a prediction with a KNORA-U model.

Running the example fits the KNORA-U model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 0
```

Listing 12.10: Example output from making a prediction with a KNORA-U model.

Now that we are familiar with using the scikit-learn API to evaluate and use KNORA models, let's look at configuring the model.

12.4 Hyperparameter Tuning for KNORA

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the KNORA model and their effect on model performance. There are many hyperparameters we can look at for KNORA, although in this case, we will look at the value of k in the k-nearest neighbor model used in the local evaluation of the models, and how to use a custom pool of classifiers. We will use the KNORA-Union as the basis for these experiments, although the choice of the specific method is arbitrary.

12.4.1 Explore k in k-Nearest Neighbors

The configuration of the k-nearest neighbors algorithm is critical to the KNORA model as it defines the scope of the neighborhood in which each ensemble is considered for selection. The k value controls the size of the neighborhood and it is important to set it to a value that is appropriate for your dataset, specifically the density of samples in the feature space. A value too small will mean that relevant examples in the training set might be excluded from the neighborhood, whereas values too large may mean that the signal is being washed out by too many examples. The code example below explores the classification accuracy of the KNORA-U algorithm with k values from 2 to 21.

```
# explore k in knn for KNORA-U dynamic ensemble selection
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 deslib.des.knora_u import KNORAU
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=10000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore k values from 2 to 21
  for n in range(2,22):
    models[str(n)] = KNORAU(k=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the scores
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the results along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 12.11: Example of tunning k in kNN for the KNORA model.

Running the example first reports the mean accuracy for each configured neighborhood size.

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

In this case, we can see that accuracy may increase slightly with the neighborhood size,

perhaps to $k = 10$, where it appears to fall off.

```
>2 0.933 (0.008)
>3 0.933 (0.010)
>4 0.935 (0.011)
>5 0.935 (0.007)
>6 0.937 (0.009)
>7 0.935 (0.011)
>8 0.937 (0.010)
>9 0.936 (0.009)
>10 0.938 (0.007)
>11 0.935 (0.010)
>12 0.936 (0.009)
>13 0.934 (0.009)
>14 0.937 (0.009)
>15 0.938 (0.009)
>16 0.935 (0.010)
>17 0.938 (0.008)
>18 0.936 (0.007)
>19 0.934 (0.007)
>20 0.935 (0.007)
>21 0.936 (0.009)
```

Listing 12.12: Example output from tunning k in kNN for the KNORA model.

A box and whisker plot is created for the distribution of accuracy scores for each configured neighborhood size.

We can see the general trend of increasing model performance and k value before reaching a plateau.

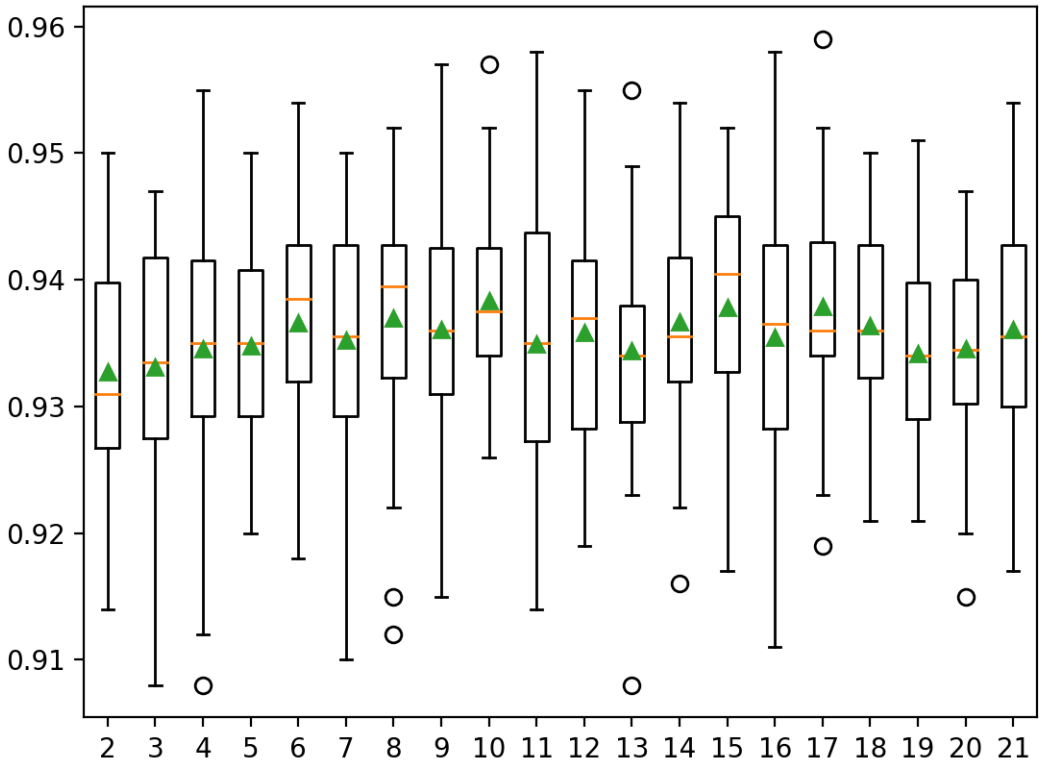

Figure 12.1: Box and Whisker Plots of Accuracy Distributions for k Values in KNORA-U.

12.4.2 Explore Algorithms for Classifier Pool

The choice of algorithms used in the pool for the KNORA is another important hyperparameter. By default, bagged decision trees are used, as it has proven to be an effective approach on a range of classification tasks. Nevertheless, a custom pool of classifiers can be considered.

> In the majority of DS publications, the pool of classifiers is generated using either well known ensemble generation methods such as Bagging, or by using heterogeneous classifiers.
>
> — *Dynamic Classifier Selection: Recent Advances And Perspectives*, 2018.

This requires first defining a list of classifier models to use and fitting each on the training dataset. Unfortunately, this means that the automatic k-fold cross-validation model evaluation methods in scikit-learn cannot be used in this case. Instead, we will use a train-test split so that we can fit the classifier pool manually on the training dataset. The list of fit classifiers can then be specified to the KNORA-Union (or KNORA-Eliminate) class via the `pool_classifiers` argument. In this case, we will use a pool that includes logistic regression, a decision tree, and a naive Bayes classifier. The complete example of evaluating the KNORA ensemble and a custom set of classifiers on the synthetic dataset is listed below.

```
# evaluate KNORA-U dynamic ensemble selection with a custom pool of algorithms
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from deslib.des.knora_u import KNORAU
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
# define classifiers to use in the pool
classifiers = [
  LogisticRegression(),
  DecisionTreeClassifier(),
  GaussianNB()]
# fit each classifier on the training set
for c in classifiers:
  c.fit(X_train, y_train)
# define the KNORA-U model
model = KNORAU(pool_classifiers=classifiers)
# fit the model
model.fit(X_train, y_train)
# make predictions on the test set
yhat = model.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Accuracy: %.3f' % (score))
```

Listing 12.13: Example of evaluating a KNORA model with a pool of custom classifiers.

Running the example first reports the mean accuracy for the model with the custom pool of classifiers.

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

In this case, we can see that the model achieved an accuracy of about 91.3 percent.

```
Accuracy: 0.913
```

Listing 12.14: Example output from evaluating a KNORA model with a pool of custom classifiers.

In order to adopt the KNORA model, it must perform better than any contributing model. Otherwise, we would simply use the contributing model that performs better. We can check this by evaluating the performance of each contributing classifier on the test set.

```
...
# evaluate contributing models
for c in classifiers:
  yhat = c.predict(X_test)
  score = accuracy_score(y_test, yhat)
  print('>%s: %.3f' % (c.__class__.__name__, score))
```

Listing 12.15: Example of evaluating standalone classifiers used in the KNORA pool.

The updated example of KNORA with a custom pool of classifiers that are also evaluated independently is listed below.

```
# evaluate KNORA-U dynamic ensemble selection with a custom pool of algorithms
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from deslib.des.knora_u import KNORAU
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
# define classifiers to use in the pool
classifiers = [
  LogisticRegression(),
  DecisionTreeClassifier(),
  GaussianNB()]
# fit each classifier on the training set
for c in classifiers:
  c.fit(X_train, y_train)
# define the KNORA-U model
model = KNORAU(pool_classifiers=classifiers)
# fit the model
model.fit(X_train, y_train)
# make predictions on the test set
yhat = model.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Accuracy: %.3f' % (score))
# evaluate contributing models
for c in classifiers:
  yhat = c.predict(X_test)
  score = accuracy_score(y_test, yhat)
  print('>%s: %.3f' % (c.__class__.__name__, score))
```

Listing 12.16: Example of evaluating standalone classifiers used in the KNORA pool.

Running the example first reports the mean accuracy for the model with the custom pool of classifiers and the accuracy of each contributing model.

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

In this case, we can see that again the `KNORAU` achieves an accuracy of about 91.3 percent, which is better than any contributing model.

```
Accuracy: 0.913
>LogisticRegression: 0.878
>DecisionTreeClassifier: 0.885
>GaussianNB: 0.873
```

Listing 12.17: Example output from evaluating standalone classifiers used in the KNORA pool.

Instead of specifying a pool of classifiers, it is also possible to specify a single ensemble algorithm from the scikit-learn library and the KNORA algorithm will automatically use the internal ensemble members as classifiers. For example, we can use a random forest ensemble with 1,000 members as the base classifiers to consider within KNORA as follows:

```
...
# define classifiers to use in the pool
pool = RandomForestClassifier(n_estimators=1000)
# fit the classifiers on the training set
pool.fit(X_train, y_train)
# define the KNORA-U model
model = KNORAU(pool_classifiers=pool)
```

Listing 12.18: Example of evaluating using random forest trees as classifiers in the KNORA pool.

Tying this together, the complete example of KNORA-U with random forest ensemble members as classifiers is listed below.

```
# evaluate KNORA-U with a random forest ensemble as the classifier pool
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from deslib.des.knora_u import KNORAU
from sklearn.ensemble import RandomForestClassifier
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
# define classifiers to use in the pool
pool = RandomForestClassifier(n_estimators=1000)
# fit the classifiers on the training set
pool.fit(X_train, y_train)
# define the KNORA-U model
model = KNORAU(pool_classifiers=pool)
# fit the model
model.fit(X_train, y_train)
# make predictions on the test set
yhat = model.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Accuracy: %.3f' % (score))
# evaluate the standalone model
yhat = pool.predict(X_test)
score = accuracy_score(y_test, yhat)
print('>%s: %.3f' % (pool.__class__.__name__, score))
```

Listing 12.19: Example of evaluating random forest classifiers used in the KNORA pool.

Running the example first reports the mean accuracy for the model with the custom pool of classifiers and the accuracy of the random forest model.

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

In this case, we can see that the KNORA model with dynamically selected ensemble members out-performs the random forest with the statically selected (full set) ensemble members.

```
Accuracy: 0.968
>RandomForestClassifier: 0.967
```

Listing 12.20: Example output from evaluating random forest classifiers used in the KNORA pool.

12.5 Further Reading

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

Papers

- *From Dynamic Classifier Selection To Dynamic Ensemble Selection*, 2008.
  https://www.sciencedirect.com/science/article/abs/pii/S0031320307004499
- *Dynamic Selection Of Classifiers: A Comprehensive Review*, 2014.
  https://www.sciencedirect.com/science/article/abs/pii/S0031320314001885
- *Dynamic Classifier Selection: Recent Advances And Perspectives*, 2018.
  https://www.sciencedirect.com/science/article/pii/S1566253517304074

Books

- *Ensemble Methods: Foundations and Algorithms*, 2012.
  https://amzn.to/32L1yWD

APIs

- Dynamic Selection Library Project, GitHub.
  https://github.com/scikit-learn-contrib/DESlib
- DESlib API Documentation.
  https://deslib.readthedocs.io/en/latest/api.html

12.6 Summary

In this tutorial, you discovered how to develop dynamic ensemble selection models in Python. Specifically, you learned:

- Dynamic ensemble selection algorithms automatically choose ensemble members when making a prediction on new data.
- How to develop and evaluate dynamic ensemble selection models for classification tasks using the scikit-learn API.
- How to explore the effect of dynamic ensemble selection model hyperparameters on classification accuracy.

Next

In the next section, we will take a closer look at the Mixture of Experts ensemble learning algorithm.

Chapter 13

Mixture of Experts Ensemble

Mixture of experts is an ensemble learning technique developed in the field of neural networks. It involves decomposing predictive modeling tasks into sub-tasks, training an expert model on each, developing a gating model that learns which expert to trust based on the input to be predicted, and combines the predictions. Although the technique was initially described using neural network experts and gating models, it can be generalized to use models of any type. As such, it shows a strong similarity to stacked generalization and belongs to the class of ensemble learning methods referred to as meta-learning. In this tutorial, you will discover the mixture of experts approach to ensemble learning. After completing this tutorial, you will know:

- An intuitive approach to ensemble learning involves dividing a task into subtasks and developing an expert on each subtask.
- Mixture of experts is an ensemble learning method that seeks to explicitly address a predictive modeling problem in terms of subtasks using expert models.
- The divide and conquer approach is related to the construction of decision trees, and the meta-learner approach is related to the stacked generalization ensemble method.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Subtasks and Experts
2. Mixture of Experts
3. Relationship With Other Techniques

13.2 Subtasks and Experts

Some predictive modeling tasks are remarkably complex, although they may be suited to a natural division into subtasks. For example, consider a one-dimensional function that has a

complex shape like an S in two dimensions. We could attempt to devise a model that models the function completely, but if we know the functional form, the S-shape, we could also divide up the problem into three parts: the curve at the top, the curve at the bottom and the line connecting the curves. This is a divide and conquer approach to problem-solving and underlies many automated approaches to predictive modeling, as well as problem-solving more broadly. This approach can also be explored as the basis for developing an ensemble learning method.

For example, we can divide the input feature space into subspaces based on some domain knowledge of the problem. A model can then be trained on each subspace of the problem, being in effect an expert on the specific subproblem. A model then learns which expert to call upon to predict new examples in the future. The subproblems may or may not overlap, and experts from similar or related subproblems may be able to contribute to the examples that are technically outside of their expertise. This approach to ensemble learning underlies a technique referred to as a mixture of experts.

13.3 Mixture of Experts

Mixture of experts, MoE or ME for short, is an ensemble learning technique that implements the idea of training experts on subtasks of a predictive modeling problem.

> In the neural network community, several researchers have examined the decomposition methodology. [...] Mixture-of-Experts (ME) methodology that decomposes the input space, such that each expert examines a different part of the space. [...] A gating network is responsible for combining the various experts.
>
> — Page 73, *Pattern Classification Using Ensemble Methods*, 2010.

There are four elements to the approach, they are:

- Division of a task into subtasks.
- Develop an expert for each subtask.
- Use a gating model to decide which expert to use.
- Pool predictions and gating model output to make a prediction.

The figure below, taken from Page 94 of the 2012 book *Ensemble Methods*, provides a helpful overview of the architectural elements of the method.

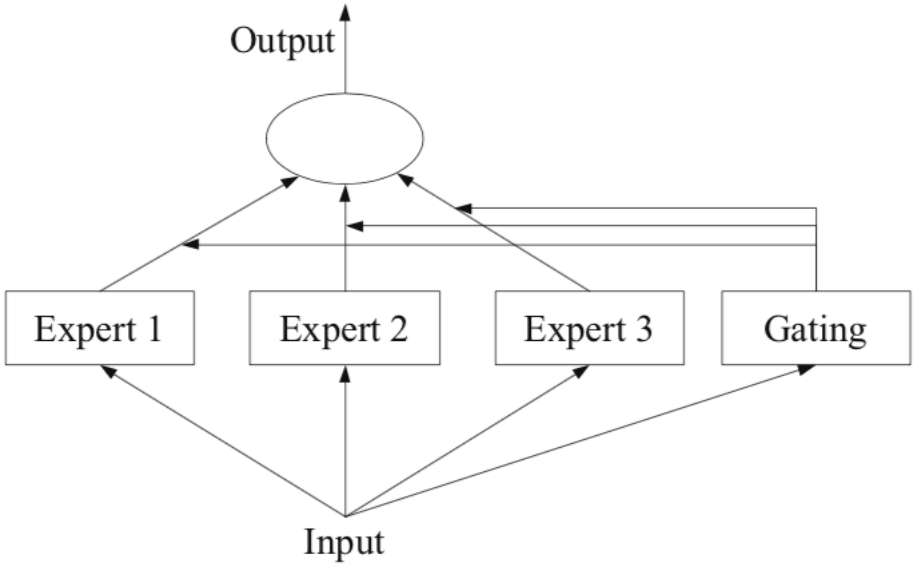

Figure 13.1: Example of a Mixture of Experts Model with Expert Members and a Gating Network Taken from: *Ensemble Methods*.

13.3.1 Subtasks

The first step is to divide the predictive modeling problem into subtasks. This often involves using domain knowledge. For example, an image could be divided into separate elements such as background, foreground, objects, colors, lines, and so on.

> ... ME works in a divide-and-conquer strategy where a complex task is broken up into several simpler and smaller subtasks, and individual learners (called experts) are trained for different subtasks.

— Page 94, *Ensemble Methods*, 2012.

For those problems where the division of the task into subtasks is not obvious, a simpler and more generic approach could be used. For example, one could imagine an approach that divides the input feature space by groups of columns or separates examples in the feature space based on distance measures, inliers, and outliers for a standard distribution, and much more.

> ... in ME, a key problem is how to find the natural division of the task and then derive the overall solution from sub-solutions.

— Page 94, *Ensemble Methods*, 2012.

13.3.2 Expert Models

Next, an expert is designed for each subtask. The mixture of experts approach was initially developed and explored within the field of artificial neural networks, so traditionally, experts themselves are neural network models used to predict a numerical value in the case of regression or a class label in the case of classification.

> It should be clear that we can "plug in" any model for the expert. For example, we can use neural networks to represent both the gating functions and the experts. The result is known as a mixture density network.
>
> — Page 344, *Machine Learning: A Probabilistic Perspective*, 2012.

Experts each receive the same input pattern (row) and make a prediction.

13.3.3 Gating Model

A model is used to interpret the predictions made by each expert and to aid in deciding which expert to trust for a given input. This is called the gating model, or the gating network, given that it is traditionally a neural network model. The gating network takes as input the input pattern that was provided to the expert models and outputs the contribution that each expert should have in making a prediction for the input.

> ... the weights determined by the gating network are dynamically assigned based on the given input, as the MoE effectively learns which portion of the feature space is learned by each ensemble member
>
> — Page 16, *Ensemble Machine Learning*, 2012.

The gating network is key to the approach and effectively the model learns to choose the type subtask for a given input and, in turn, the expert to trust to make a strong prediction.

> Mixture-of-experts can also be seen as a classifier selection algorithm, where individual classifiers are trained to become experts in some portion of the feature space.
>
> — Page 16, *Ensemble Machine Learning*, 2012.

When neural network models are used, the gating network and the experts are trained together such that the gating network learns when to trust each expert to make a prediction. This training procedure was traditionally implemented using expectation maximization (EM). The gating network might have a softmax output that gives a probability-like confidence score for each expert.

> In general, the training procedure tries to achieve two goals: for given experts, to find the optimal gating function; for a given gating function, to train the experts on the distribution specified by the gating function.
>
> — Page 95, *Ensemble Machine Learning*, 2012.

13.3.4 Pooling Method

Finally, the mixture of expert models must make a prediction, and this is achieved using a pooling or aggregation mechanism. This might be as simple as selecting the expert with the largest output or confidence provided by the gating network. Alternatively, a weighted sum prediction could be made that explicitly combines the predictions made by each expert and the confidence estimated by the gating network. You might imagine other approaches to making effective use of the predictions and gating network output.

> The pooling/combining system may then choose a single classifier with the highest weight, or calculate a weighted sum of the classifier outputs for each class, and pick the class that receives the highest weighted sum.
>
> — Page 16, *Ensemble Machine Learning*, 2012.

13.4 Relationship With Other Techniques

The mixture of experts method is less popular today, perhaps because it was described in the field of neural networks. Nevertheless, more than 25 years of advancements and exploration of the technique have occurred and you can see a great summary in the 2012 paper *Twenty Years of Mixture of Experts*. Importantly, I'd recommend considering the broader intent of the technique and explore how you might use it on your own predictive modeling problems.

For example:

- Are there obvious or systematic ways that you can divide your predictive modeling problem into subtasks?
- Are there specialized methods that you can train on each subtask?
- Consider developing a model that predicts the confidence of each expert model.

13.4.1 Mixture of Experts and Decision Trees

We can also see a relationship between a mixture of experts to Classification And Regression Trees, often referred to as CART. Decision trees are fit using a divide and conquer approach to the feature space. Each split is chosen as a constant value for an input feature and each sub-tree can be considered a submodel.

> Mixture of experts was mostly studied in the neural networks community. In this thread, researchers generally consider a divide-and-conquer strategy, try to learn a mixture of parametric models jointly and use combining rules to get an overall solution.
>
> — Page 16, *Ensemble Methods*, 2012.

We could take a similar recursive decomposition approach to decomposing the predictive modeling task into subproblems when designing the mixture of experts. This is generally referred to as a hierarchical mixture of experts.

> The hierarchical mixtures of experts (HME) procedure can be viewed as a variant of tree-based methods. The main difference is that the tree splits are not hard decisions but rather soft probabilistic ones.
>
> — Page 329, *The Elements of Statistical Learning*, 2016.

Unlike decision trees, the division of the task into subtasks is often explicit and top-down. Also, unlike a decision tree, the mixture of experts attempts to survey all of the expert submodels rather than a single model.

> There are other differences between HMEs and the CART implementation of trees. In an HME, a linear (or logistic regression) model is fit in each terminal node, instead of a constant as in CART. The splits can be multiway, not just binary, and the splits are probabilistic functions of a linear combination of inputs, rather than a single input as in the standard use of CART.
>
> — Page 329, *The Elements of Statistical Learning*, 2016.

Nevertheless, these differences might inspire variations on the approach for a given predictive modeling problem. For example:

- Consider automatic or general approaches to dividing the feature space or problem into subtasks to help to broaden the suitability of the method.
- Consider exploring both combination methods that trust the best expert, as well as methods that seek a weighted consensus across experts.

13.4.2 Mixture of Experts and Stacking

The application of the technique does not have to be limited to neural network models and a range of standard machine learning techniques can be used in place seeking a similar end. In this way, the mixture of experts method belongs to a broader class of ensemble learning methods that would also include stacked generalization, known as stacking. Like a mixture of experts, stacking trains a diverse ensemble of machine learning models and then learns a higher-order model to best combine the predictions. We might refer to this class of ensemble learning methods as meta-learning models. That is models that attempt to learn from the output or learn how to best combine the output of other lower-level models.

> Meta-learning is a process of learning from learners (classifiers). [...] In order to induce a meta classifier, first the base classifiers are trained (stage one), and then the Meta classifier (second stage).
>
> — Page 82, *Pattern Classification Using Ensemble Methods*, 2010.

Unlike a mixture of experts, stacking models are often all fit on the same training dataset, e.g. no decomposition of the task into subtasks. And also unlike a mixture of experts, the higher-level model that combines the predictions from the lower-level models typically does not receive the input pattern provided to the lower-level models and instead takes as input the predictions from each lower-level model.

> Meta-learning methods are best suited for cases in which certain classifiers consistently correctly classify, or consistently misclassify, certain instances.
>
> — Page 82, *Pattern Classification Using Ensemble Methods*, 2010.

Nevertheless, there is no reason why hybrid stacking and mixture of expert models cannot be developed that may perform better than either approach in isolation on a given predictive modeling problem.

For example:

- Consider treating the lower-level models in stacking as experts trained on different perspectives of the training data. Perhaps this could involve using a softer approach to decomposing the problem into subproblems where different data transforms or feature selection methods are used for each model.
- Consider providing the input pattern to the meta-model in stacking in an effort to make the weighting or contribution of lower-level models conditional on the specific context of the prediction.

13.5 Further Reading

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

Papers

- *Twenty Years of Mixture of Experts*, 2012.
  https://ieeexplore.ieee.org/abstract/document/6215056

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5
- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/3frGM1A
- *The Elements of Statistical Learning*, 2016.
  https://amzn.to/31mzA31
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2YrVLmp

- *Neural Networks for Pattern Recognition*, 1995.
  https://amzn.to/3fWgcOh
- *Deep Learning*, 2016.
  https://amzn.to/3ds4KZ8

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Mixture of experts, Wikipedia.
  https://en.wikipedia.org/wiki/Mixture_of_experts

13.6 Summary

In this tutorial, you discovered mixture of experts approach to ensemble learning. Specifically, you learned:

- An intuitive approach to ensemble learning involves dividing a task into subtasks and developing an expert on each subtask.
- Mixture of experts is an ensemble learning method that seeks to explicitly address a predictive modeling problem in terms of subtasks using expert models.
- The divide and conquer approach is related to the construction of decision trees, and the meta-learner approach is related to the stacked generalization ensemble method.

Next

This was the final tutorial in this part, in the next part will take a closer look at bagging ensemble algorithms.

Part V

Bagging

Chapter 14

Bagged Decision Trees Ensemble

Bagging is an ensemble machine learning algorithm that combines the predictions from many decision trees. It is also easy to implement given that it has few key hyperparameters and sensible heuristics for configuring these hyperparameters. Bagging performs well in general and provides the basis for a whole field of ensemble of decision tree algorithms such as the popular random forest and extra trees ensemble algorithms, as well as the lesser-known Pasting, Random Subspaces, and Random Patches ensemble algorithms. In this tutorial, you will discover how to develop Bagging ensembles for classification and regression. After completing this tutorial, you will know:

- Bagging ensemble is an ensemble created from decision trees fit on different samples of a dataset.
- How to use the Bagging ensemble for classification and regression with scikit-learn.
- How to explore the effect of Bagging model hyperparameters on model performance.

Let's get started.

14.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Bagging Ensemble Algorithm
2. Evaluate Bagging Ensembles
3. Bagging Hyperparameters
4. Bagging Extensions
5. Common Questions

14.2 Bagging Ensemble Algorithm

Bootstrap Aggregation, or Bagging for short, is an ensemble machine learning algorithm. Specifically, it is an ensemble of decision tree models, although the bagging technique can also be used to combine the predictions of other types of models. As its name suggests, bootstrap aggregation is based on the idea of the *bootstrap* sample. A bootstrap sample is a sample of a dataset with replacement. Replacement means that a sample drawn from the dataset is replaced, allowing it to be selected again and perhaps multiple times in the new sample. This means that the sample may have duplicate examples from the original dataset. The bootstrap sampling technique is used to estimate a population statistic from a small data sample. This is achieved by drawing multiple bootstrap samples, calculating the statistic on each, and reporting the mean statistic across all samples.

An example of using bootstrap sampling would be estimating the population mean from a small dataset. Multiple bootstrap samples are drawn from the dataset, the mean calculated on each, then the mean of the estimated means is reported as an estimate of the population mean. Surprisingly, the bootstrap method provides a robust and accurate approach to estimating statistical quantities compared to a single estimate on the original dataset.

This same approach can be used to create an ensemble of decision tree models. This is achieved by drawing multiple bootstrap samples from the training dataset and fitting a decision tree on each. The predictions from the decision trees are then combined to provide a more robust and accurate prediction than a single decision tree (typically, but not always).

> Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. [...] The multiple versions are formed by making bootstrap replicates of the learning set and using these as new learning sets.
>
> — *Bagging Predictors*, 1996.

Predictions are made for regression problems by averaging the prediction across the decision trees. Predictions are made for classification problems by taking the majority vote prediction for the classes from across the predictions made by the decision trees. The bagged decision trees are effective because each decision tree is fit on a slightly different training dataset, which in turn allows each tree to have minor differences and make slightly different skillful predictions. Technically, we say that the method is effective because the trees have a low correlation between predictions and, in turn, prediction errors.

Decision trees, specifically unpruned decision trees, are used as they slightly overfit the training data and have a high variance. Other high-variance machine learning algorithms can be used, such as a k-nearest neighbors algorithm with a low k value, although decision trees have proven to be the most effective.

> If perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy.
>
> — *Bagging Predictors*, 1996.

Bagging does not always offer an improvement. For low-variance models that already perform well, bagging can result in a decrease in model performance.

> The evidence, both experimental and theoretical, is that bagging can push a good but unstable procedure a significant step towards optimality. On the other hand, it can slightly degrade the performance of stable procedures.
>
> — *Bagging Predictors*, 1996.

14.3 Evaluate Bagging Ensembles

The scikit-learn Python machine learning library provides an implementation of Bagging ensembles for machine learning via the `BaggingRegressor` and `BaggingClassifier` classes. Both models operate the same way and take the same arguments that influence how the decision trees are created. Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model. When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions. Let's take a look at how to develop a Bagging ensemble for both classification and regression.

14.3.1 Bagging for Classification

In this section, we will look at using Bagging for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=5)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 14.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 14.2: Example output from creating the synthetic classification dataset.

Next, we can evaluate a Bagging algorithm on this dataset. We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate bagging algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
```

```
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the model
model = BaggingClassifier()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the results
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 14.3: Example of evaluating bagging on a classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Bagging ensemble with default hyperparameters achieves a classification accuracy of about 85 percent on this synthetic dataset.

```
Mean Accuracy: 0.856 (0.037)
```

Listing 14.4: Example output from evaluating bagging on a classification dataset.

We can also use the Bagging model as a final model and make predictions for classification. First, the Bagging ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make predictions using bagging for classification
from sklearn.datasets import make_classification
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the model
model = BaggingClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [-4.7705504, -1.88685058, -0.96057964, 2.53850317, -6.5843005, 3.45711663,
    -7.46225013, 2.01338213, -0.45086384, -1.89314931, -2.90675203, -0.21214568,
    -0.9623956, 3.93862591, 0.06276375, 0.33964269, 4.0835676, 1.31423977, -2.17983117,
    3.1047287]
yhat = model.predict([row])
# summarize the prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 14.5: Example of using bagging for making a prediction on a classification dataset.

Running the example fits the Bagging ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 14.6: Example output from using bagging for making a prediction on a classification dataset.

Now that we are familiar with using Bagging for classification, let's look at the API for regression.

14.3.2 Bagging for Regression

In this section, we will look at using Bagging for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=5)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 14.7: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 14.8: Example output from creating the synthetic regression dataset.

Next, we can evaluate a Bagging algorithm on this dataset. As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate bagging ensemble for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.ensemble import BaggingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=5)
# define the model
model = BaggingRegressor()
```

```
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the results
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 14.9: Example of evaluating bagging on a regression dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see that the Bagging ensemble with default hyperparameters achieves a MAE of about 100.

```
MAE: -101.133 (9.757)
```

Listing 14.10: Example output from evaluating bagging on a regression dataset.

We can also use the Bagging model as a final model and make predictions for regression. First, the Bagging ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# bagging ensemble for making predictions for regression
from sklearn.datasets import make_regression
from sklearn.ensemble import BaggingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=5)
# define the model
model = BaggingRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.88950817, -0.93540416, 0.08392824, 0.26438806, -0.52828711, -1.21102238,
    -0.4499934, 1.47392391, -0.19737726, -0.22252503, 0.02307668, 0.26953276, 0.03572757,
    -0.51606983, -0.39937452, 1.8121736, -0.00775917, -0.02514283, -0.76089365, 1.58692212]
yhat = model.predict([row])
# summarize the prediction
print('Prediction: %d' % yhat[0])
```

Listing 14.11: Example of using bagging for making a prediction on a regression dataset.

Running the example fits the Bagging ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: -134
```

Listing 14.12: Example output from using bagging for making a prediction on a regression dataset.

Now that we are familiar with using the scikit-learn API to evaluate and use Bagging ensembles, let's look at configuring the model.

14.4 Bagging Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the Bagging ensemble and their effect on model performance.

14.4.1 Explore Number of Trees

An important hyperparameter for the Bagging algorithm is the number of decision trees used in the ensemble. Typically, the number of trees is increased until the model performance stabilizes. Intuition might suggest that more trees will lead to overfitting, although this is not the case. Bagging and related ensembles of decision trees algorithms (like random forest) appear to be somewhat immune to overfitting the training dataset given the stochastic nature of the learning algorithm. The number of trees can be set via the `n_estimators` argument and defaults to 100. The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore bagging ensemble number of trees effect on performance
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.ensemble import BaggingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=5)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define number of trees to consider
  n_trees = [10, 50, 100, 500, 500, 1000, 5000]
  for n in n_trees:
    models[str(n)] = BaggingClassifier(n_estimators=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
```

```
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 14.13: Example of evaluating the effect of the number of trees in the bagging ensemble.

Running the example first reports the mean accuracy for each configured number of decision trees.

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

In this case, we can see that performance improves on this dataset until about 100 trees and remains flat after that.

```
>10 0.855 (0.037)
>50 0.876 (0.035)
>100 0.882 (0.037)
>500 0.885 (0.041)
>1000 0.885 (0.037)
>5000 0.885 (0.038)
```

Listing 14.14: Example output from evaluating the effect of the number of trees in the bagging ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees. We can see the general trend of no further improvement beyond about 100 trees.

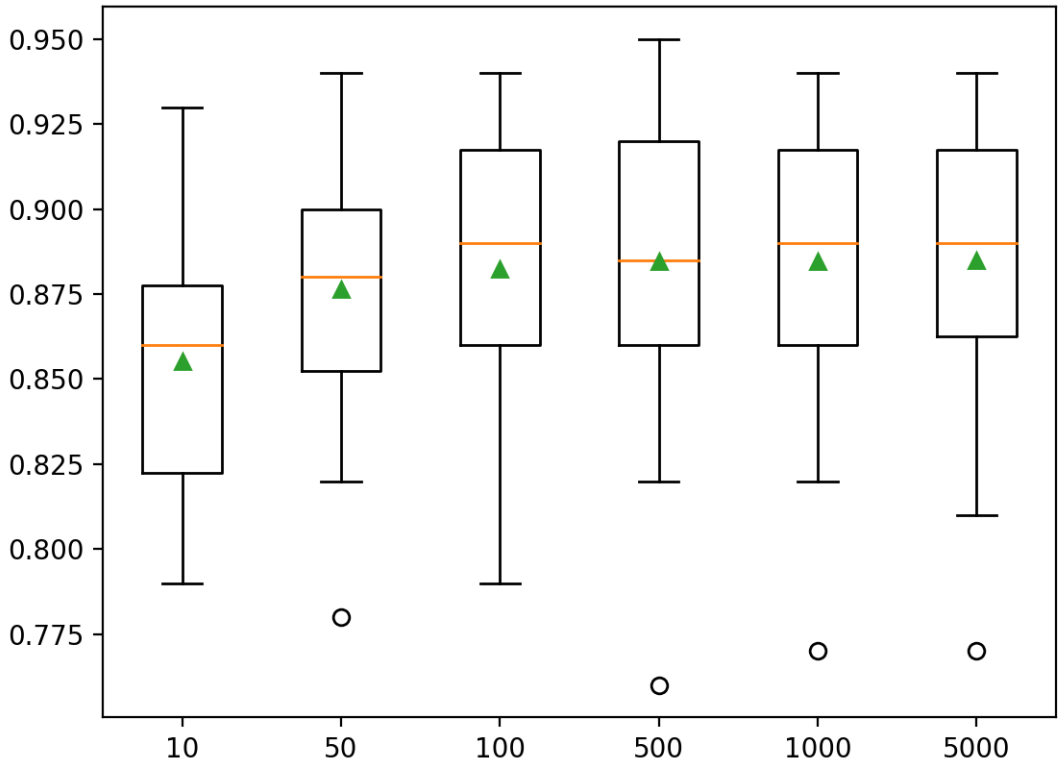

Figure 14.1: Box Plot of Bagging Ensemble Size vs. Classification Accuracy.

14.4.2 Explore Number of Samples

The size of the bootstrap sample can also be varied. The default is to create a bootstrap sample that has the same number of examples as the original dataset. Using a smaller dataset can increase the variance of the resulting decision trees and could result in better overall performance. The number of samples used to fit each decision tree is set via the `max_samples` argument. The example below explores different sized samples as a ratio of the original dataset from 10 percent to 100 percent (the default).

```
# explore bagging ensemble number of samples effect on performance
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
```

```
      n_redundant=5, random_state=5)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore ratios from 10% to 100% in 10% increments
  for i in arange(0.1, 1.1, 0.1):
    key = '%.1f' % i
    models[key] = BaggingClassifier(max_samples=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 14.15: Example of evaluating the effect of the number of samples in the bagging ensemble.

Running the example first reports the mean accuracy for each sample set size.

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

In this case, the results suggest that performance generally improves with an increase in the sample size, highlighting that the default of 100 percent the size of the training dataset is sensible. It might also be interesting to explore a smaller sample size with a corresponding increase in the number of trees in an effort to reduce the variance of the individual models.

```
>0.1 0.810 (0.036)
>0.2 0.836 (0.044)
>0.3 0.844 (0.043)
>0.4 0.843 (0.041)
```

```
>0.5 0.852 (0.034)
>0.6 0.855 (0.042)
>0.7 0.858 (0.042)
>0.8 0.861 (0.033)
>0.9 0.866 (0.041)
>1.0 0.864 (0.042)
```

Listing 14.16: Example output from evaluating the effect of the number of samples in the bagging ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each sample size. We see a general trend of increasing accuracy with sample size.

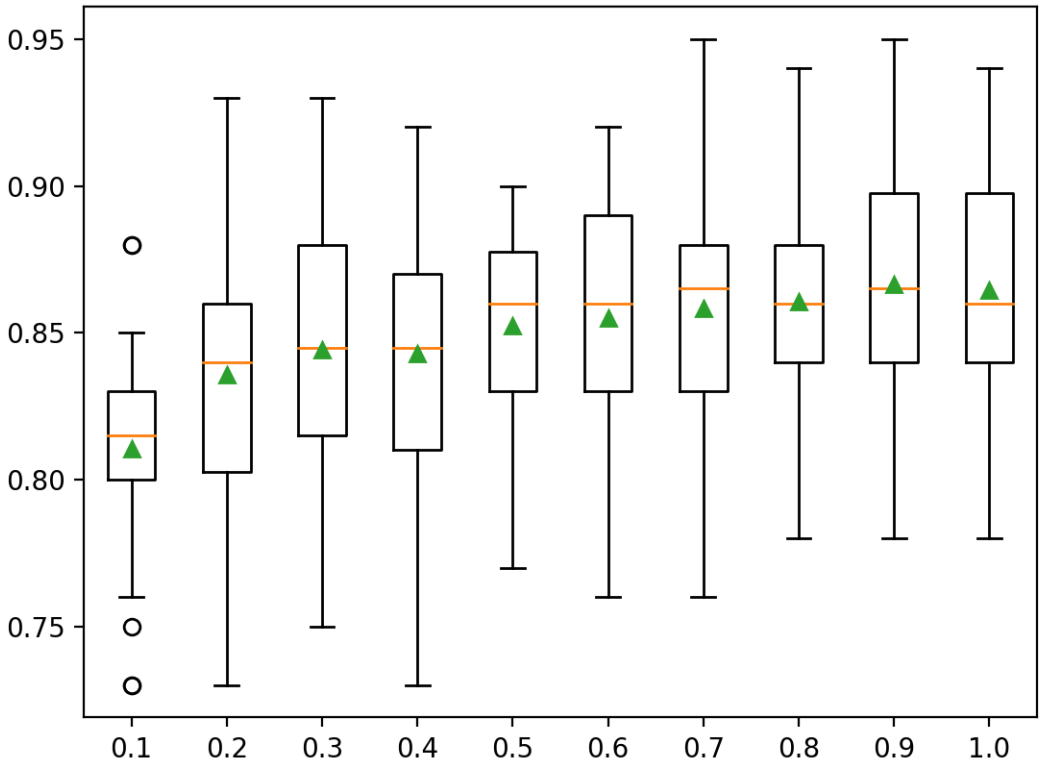

Figure 14.2: Box Plot of Bagging Sample Size vs. Classification Accuracy.

14.4.3 Explore Alternate Algorithm

Decision trees are the most common algorithm used in a bagging ensemble. The reason for this is that they are easy to configure to have a high variance and because they perform well in general. Other algorithms can be used with bagging and must be configured to have a modestly high variance. One example is the k-nearest neighbors algorithm where the k value can be set to a low value. The algorithm used in the ensemble is specified via the `base_estimator` argument and must be set to an instance of the algorithm and algorithm configuration to use.

The example below demonstrates using a `KNeighborsClassifier` as the base algorithm used in the bagging ensemble. Here, the algorithm is used with default hyperparameters where k is set to 5.

```
# evaluate bagging with knn algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the model
model = BaggingClassifier(base_estimator=KNeighborsClassifier())
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the results
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 14.17: Example of evaluating the effect of changing the base algorithm in the bagging ensemble.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Bagging ensemble with KNN and default hyperparameters achieves a classification accuracy of about 88 percent on this synthetic dataset.

```
Mean Accuracy: 0.888 (0.036)
```

Listing 14.18: Example output from evaluating the effect of changing the base algorithm in the bagging ensemble.

We can test different values of k to find the right balance of model variance to achieve good performance as a bagged ensemble. The below example tests bagged KNN models with k values between 1 and 20.

```
# explore bagging ensemble k for knn effect on performance
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.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from matplotlib import pyplot

# get the dataset
```

```
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=5)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # evaluate k values from 1 to 20
  for i in range(1,21):
    # define the base model
    base = KNeighborsClassifier(n_neighbors=i)
    # define the ensemble model
    models[str(i)] = BaggingClassifier(base_estimator=base)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 14.19: Example of evaluating the effect of changing the configuration of KNN as the base algorithm in the bagging ensemble.

Running the example first reports the mean accuracy for each k value.

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

In this case, the results suggest a small k value such as two to four results in the best mean accuracy when used in a bagging ensemble.

```
>1 0.884 (0.025)
>2 0.890 (0.029)
```

```
>3 0.886 (0.035)
>4 0.887 (0.033)
>5 0.878 (0.037)
>6 0.879 (0.042)
>7 0.877 (0.037)
>8 0.877 (0.036)
>9 0.871 (0.034)
>10 0.877 (0.033)
>11 0.876 (0.037)
>12 0.877 (0.030)
>13 0.874 (0.034)
>14 0.871 (0.039)
>15 0.875 (0.034)
>16 0.877 (0.033)
>17 0.872 (0.034)
>18 0.873 (0.036)
>19 0.876 (0.034)
>20 0.876 (0.037)
```

Listing 14.20: Example output from evaluating the effect of changing the configuration of KNN as the base algorithm in the bagging ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each k value. We see a general trend of increasing accuracy with sample size in the beginning, then a modest decrease in performance as the variance of the individual KNN models used in the ensemble is increased with larger k values.

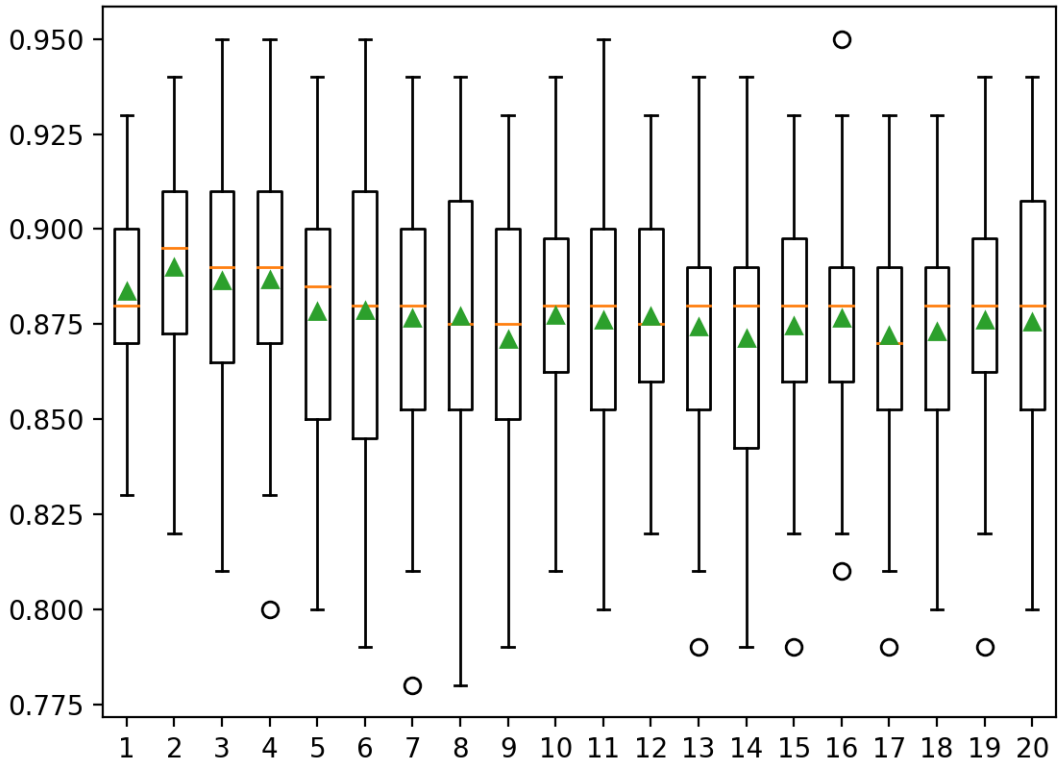

Figure 14.3: Box Plot of Bagging KNN Number of Neighbors vs. Classification Accuracy.

14.5 Bagging Extensions

There are many modifications and extensions to the bagging algorithm in an effort to improve the performance of the approach. Perhaps the most famous is the random forest algorithm. There is a number of less famous, although still effective, extensions to bagging that may be interesting to investigate. This section demonstrates some of these approaches, such as pasting ensemble, random subspace ensemble, and the random patches ensemble. We are not comparing the results of these extensions on the dataset , but rather providing working examples of how to use each technique that you can copy-paste and try with your own dataset.

14.5.1 Pasting Ensemble

The Pasting Ensemble is an extension to bagging that involves fitting ensemble members based on random samples of the training dataset instead of bootstrap samples. The approach is designed to use smaller sample sizes than the training dataset in cases where the training dataset does not fit into memory.

> The procedure takes small pieces of the data, grows a predictor on each small piece and then pastes these predictors together. A version is given that scales up to terabyte data sets. The methods are also applicable to on-line learning.

— *Pasting Small Votes for Classification in Large Databases and On-Line*, 1999.

The example below demonstrates the Pasting ensemble by setting the `bootstrap` argument to `False` and setting the number of samples used in the training dataset via `max_samples` to a modest value, in this case, 50 percent of the training dataset size.

```
# evaluate pasting ensemble algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the model
model = BaggingClassifier(bootstrap=False, max_samples=0.5)
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the results
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 14.21: Example of evaluating a pasting ensemble.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Pasting ensemble achieves a classification accuracy of about 84 percent on this dataset.

```
Mean Accuracy: 0.848 (0.039)
```

Listing 14.22: Example output from evaluating a pasting ensemble.

14.5.2 Random Subspace Ensemble

A Random Subspace Ensemble is an extension to bagging that involves fitting ensemble members based on datasets constructed from random subsets of the features in the training dataset. It is similar to the random forest except the data samples are random rather than a bootstrap sample and the subset of features is selected for the entire decision tree rather than at each split point in the tree.

> The classifier consists of multiple trees constructed systematically by pseudorandomly selecting subsets of components of the feature vector, that is, trees constructed in randomly chosen subspaces.
>
> — *The Random Subspace Method For Constructing Decision Forests*, 1998.

A worked example of the Random Subspace Ensemble is explored next in Chapter 15.

14.5.3 Random Patches Ensemble

The Random Patches Ensemble is an extension to bagging that involves fitting ensemble members based on datasets constructed from random subsets of rows (samples) and columns (features) of the training dataset. It does not use bootstrap samples and might be considered an ensemble that combines both the random sampling of the dataset of the Pasting ensemble and the random sampling of features of the Random Subspace ensemble.

> We investigate a very simple, yet effective, ensemble framework that builds each individual model of the ensemble from a random patch of data obtained by drawing random subsets of both instances and features from the whole dataset.
>
> — *Ensembles on Random Patches*, 2012.

The example below demonstrates the Random Patches Ensemble with decision trees created from a random sample of the training dataset limited to 50 percent of the size of the training dataset, and with a random subset of 10 features.

```
# evaluate random patches ensemble algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the model
model = BaggingClassifier(bootstrap=False, max_features=10, max_samples=0.5)
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model and collect the results
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 14.23: Example of evaluating a random patches ensemble.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Random Patches Ensemble achieves a classification accuracy of about 84 percent on this dataset.

```
Mean Accuracy: 0.845 (0.036)
```

Listing 14.24: Example output from evaluating a random patches ensemble.

14.6 Common Questions

In this section we will take a closer look at some common sticking points you may have with the bagging ensemble procedure.

Q. What algorithm should be used in the ensemble?

The algorithm should have a moderate variance, meaning it is moderately dependent upon the specific training data. A decision tree, often unpruned, is the default model to use because it works well in practice. Other algorithms can be used as long as they are configured to have a moderate variance.

> ... it is well known that Bagging should be used with unstable learners, and generally, the more unstable, the larger the performance improvement.
>
> — Page 52, *Ensemble Methods*, 2012.

Q. How many ensemble members should be used?

The performance of the model will converge with an increase of the number of decision trees to a point, then remain level. Therefore, keep increasing the number of trees until the performance stabilizes on your dataset.

> ... the performance of Bagging converges as the ensemble size, i.e., the number of base learners, grows large ...
>
> — Page 52, *Ensemble Methods*, 2012.

Q. Won't the ensemble overfit with too many trees?

No. Bagging ensembles are very unlikely to overfit in general.

Q. How large should the bootstrap sample be?

It is good practice to make the bootstrap sample as large as the original dataset size. That is 100% the size or an equal number of rows as the original dataset.

Q. What problems are well suited to bagging?

Generally, bagging is well suited to problems with small or modest sized datasets. But this is a rough guide. If you're unsure, try it and see.

> Bagging is best suited for problems with relatively small available training datasets.
>
> — Page 12, *Ensemble Machine Learning*, 2012.

14.7 Further Reading

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

Papers

- *Bagging predictors*, 1996.
  https://link.springer.com/article/10.1007/BF00058655
- *Pasting Small Votes for Classification in Large Databases and On-Line*, 1999.
  https://link.springer.com/article/10.1023/A:1007563306331
- *The Random Subspace Method For Constructing Decision Forests*, 1998.
  https://ieeexplore.ieee.org/abstract/document/709601
- *Ensembles on Random Patches*, 2012.
  https://link.springer.com/chapter/10.1007/978-3-642-33460-3_28

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

APIs

- `sklearn.ensemble.BaggingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html
- `sklearn.ensemble.BaggingRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingRegressor.html

Articles

- Bootstrap aggregating, Wikipedia.
  https://en.wikipedia.org/wiki/Bootstrap_aggregating

14.8 Summary

In this tutorial, you discovered how to develop Bagging ensembles for classification and regression. Specifically, you learned:

- Bagging ensemble is an ensemble created from decision trees fit on different samples of a dataset.
- How to use the Bagging ensemble for classification and regression with scikit-learn.
- How to explore the effect of Bagging model hyperparameters on model performance.

Next

In the next section, we will take a closer look at an extension to Bagging called Random Subspace Ensembles.

Chapter 15

Random Subspace Ensemble

Random Subspace Ensemble is a machine learning algorithm that combines the predictions from multiple decision trees trained on different subsets of columns in the training dataset. Randomly varying the columns used to train each contributing member of the ensemble has the effect of introducing diversity into the ensemble and, in turn, can lift performance over using a single decision tree. It is related to other ensembles of decision trees such as bootstrap aggregation (bagging) that creates trees using different samples of rows from the training dataset, and random forest that combines ideas from bagging and the random subspace ensemble. Although decision trees are often used, the general random subspace method can be used with any machine learning model whose performance varies meaningfully with the choice of input features. In this tutorial, you will discover how to develop random subspace ensembles for classification and regression. After completing this tutorial, you will know:

- Random subspace ensembles are created from decision trees fit on different samples of features (columns) in the training dataset.
- How to use the random subspace ensemble for classification and regression with scikit-learn.
- How to explore the effect of random subspace model hyperparameters on model performance.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Random Subspace Ensemble
2. Evaluate Random Subspace Ensembles
3. Random Subspace Ensemble Hyperparameters

15.2 Random Subspace Ensemble

A predictive modeling problem consists of one or more input variables and a target variable. A variable is a column in the data and is also often referred to as a feature. We can consider all input features together as defining an n-dimensional vector space, where n is the number of input features and each example (input row of data) is a point in the feature space. This is a common conceptualization in machine learning and as input feature spaces become larger, the distance between points in the space increases, known generally as the curse of dimensionality. A subset of input features can, therefore, be thought of as a subset of the input feature space, or a subspace.

Selecting features is a way of defining a subspace of the input feature space. For example, feature selection refers to an attempt to reduce the number of dimensions of the input feature space by selecting a subset of features to keep or a subset of features to delete, often based on their relationship to the target variable. Alternatively, we can select random subsets of input features to define random subspaces. This can be used as the basis for an ensemble learning algorithm, where a model can be fit on each random subspace of features. This is referred to as a random subspace ensemble or the random subspace method.

> The training data is usually described by a set of features. Different subsets of features, or called subspaces, provide different views on the data. Therefore, individual learners trained from different subspaces are usually diverse.
>
> — Page 116, *Ensemble Methods*, 2012.

It was proposed by Tin Kam Ho in the 1998 paper titled *The Random Subspace Method For Constructing Decision Forests* where a decision tree is fit on each random subspace. More generally, it is a diversity technique for ensemble learning that belongs to a class of methods that change the training dataset for each model in the attempt to reduce the correlation between the predictions of the models in the ensemble. The procedure is as simple as selecting a random subset of input features (columns) for each model in the ensemble and fitting the model on the entire training dataset. It can be augmented with additional changes, such as using a bootstrap or random sample of the rows in training dataset.

> The classifier consists of multiple trees constructed systematically by pseudorandomly selecting subsets of components of the feature vector, that is, trees constructed in randomly chosen subspaces.
>
> — *The Random Subspace Method For Constructing Decision Forests*, 1998.

As such, the random subspace ensemble is related to bootstrap aggregation (bagging) that introduces diversity by training each model, often a decision tree, on a different random sample of the training dataset, with replacement (e.g. the bootstrap sampling method). The random forest ensemble may also be considered a hybrid of both the bagging and random subset ensemble methods.

> Algorithms that use different feature subsets are commonly referred to as random subspace methods ...

— Page 21, *Ensemble Machine Learning*, 2012.

The random subspace method can be used with any machine learning algorithm, although it is well suited to models that are sensitive to large changes to the input features, such as decision trees and k-nearest neighbors. It is appropriate for datasets that have a large number of input features, as it can result in good performance with good efficiency. If the dataset contains many irrelevant input features, it may be better to use feature selection as a data preparation technique as the prevalence of irrelevant features in subspaces may hurt the performance of the ensemble.

> For data with a lot of redundant features, training a learner in a subspace will be not only effective but also efficient.

— Page 116, *Ensemble Methods*, 2012.

Now that we are familiar with the random subspace ensemble, let's explore how we can implement the approach.

15.3 Evaluate Random Subspace Ensembles

We can implement the random subspace ensemble using bagging in scikit-learn. Bagging is provided via the `BaggingRegressor` and `BaggingClassifier` classes. We can configure bagging to be a random subspace ensemble by setting the `bootstrap` argument to `False` to turn off sampling of the training dataset rows and setting the maximum number of features to a given value via the `max_features` argument. The default model for bagging is a decision tree, but it can be changed to any model we like. We can demonstrate using bagging to implement a random subspace ensemble with decision trees for classification and regression.

15.3.1 Random Subspace Ensemble for Classification

In this section, we will look at developing a random subspace ensemble using bagging for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=5)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 15.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 15.2: Example output from creating the synthetic classification dataset.

Next, we can configure a bagging model to be a random subspace ensemble for decision trees on this dataset. Each model will be fit on a random subspace of 10 input features, chosen arbitrarily.

```
...
# define the random subspace ensemble model
model = BaggingClassifier(bootstrap=False, max_features=10)
```

Listing 15.3: Example of defining a random subspace ensemble for classification.

We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate random subspace ensemble via bagging for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the random subspace ensemble model
model = BaggingClassifier(bootstrap=False, max_features=10)
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 15.4: Example of evaluating a random subspace ensemble on a classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the random subspace ensemble with default hyperparameters achieves a classification accuracy of about 85.4 percent on this synthetic dataset.

```
Mean Accuracy: 0.854 (0.039)
```

Listing 15.5: Example output from evaluating a random subspace ensemble on a classification dataset.

We can also use the random subspace ensemble model as a final model and make predictions for classification. First, the ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make predictions using random subspace ensemble via bagging for classification
from sklearn.datasets import make_classification
```

```
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the model
model = BaggingClassifier(bootstrap=False, max_features=10)
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [-4.7705504, -1.88685058, -0.96057964, 2.53850317, -6.5843005, 3.45711663,
    -7.46225013, 2.01338213, -0.45086384, -1.89314931, -2.90675203, -0.21214568,
    -0.9623956, 3.93862591, 0.06276375, 0.33964269, 4.0835676, 1.31423977, -2.17983117,
    3.1047287]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 15.6: Example of making a prediction with a random subspace ensemble on a classification dataset.

Running the example fits the random subspace ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 15.7: Example output from making a prediction with a random subspace ensemble on a classification dataset.

Now that we are familiar with using bagging for classification, let's look at the API for regression.

15.3.2 Random Subspace Ensemble for Regression

In this section, we will look at using bagging for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=5)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 15.8: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 15.9: Example output from creating the synthetic regression dataset.

Next, we can evaluate a random subspace ensemble via bagging on this dataset. As before, we must configure bagging to use all rows of the training dataset and specify the number of input features to randomly select.

```
...
# define the model
model = BaggingRegressor(bootstrap=False, max_features=10)
```

Listing 15.10: Example of defining a random subspace ensemble for regression.

As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate random subspace ensemble via bagging for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.ensemble import BaggingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=5)
# define the model
model = BaggingRegressor(bootstrap=False, max_features=10)
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 15.11: Example of evaluating a random subspace ensemble on a regression dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see that the bagging ensemble with default hyperparameters achieves a MAE of about 114.

```
MAE: -114.630 (10.920)
```

Listing 15.12: Example output from evaluating a random subspace ensemble on a regression dataset.

We can also use the random subspace ensemble model as a final model and make predictions for regression. First, the ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# random subspace ensemble via bagging for making predictions for regression
from sklearn.datasets import make_regression
from sklearn.ensemble import BaggingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=5)
# define the model
model = BaggingRegressor(bootstrap=False, max_features=10)
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.88950817, -0.93540416, 0.08392824, 0.26438806, -0.52828711, -1.21102238,
    -0.4499934, 1.47392391, -0.19737726, -0.22252503, 0.02307668, 0.26953276, 0.03572757,
    -0.51606983, -0.39937452, 1.8121736, -0.00775917, -0.02514283, -0.76089365, 1.58692212]
yhat = model.predict([row])
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Listing 15.13: Example of making a prediction with a random subspace ensemble on a regression dataset.

Running the example fits the random subspace ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: -157
```

Listing 15.14: Example output from making a prediction with a random subspace ensemble on a regression dataset.

Now that we are familiar with using the scikit-learn API to evaluate and use random subspace ensembles, let's look at configuring the model.

15.4 Random Subspace Ensemble Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the random subspace ensemble and their effect on model performance.

15.4.1 Explore Number of Trees

An important hyperparameter for the random subspace method is the number of decision trees used in the ensemble. More trees will stabilize the variance of the model, countering the effect of the number of features selected by each tree that introduces diversity. The number of trees can be set via the `n_estimators` argument and defaults to 10. The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore random subspace ensemble number of trees effect on performance
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.ensemble import BaggingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=5)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define number of trees to consider
  n_trees = [10, 50, 100, 500, 1000, 5000]
  for n in n_trees:
    models[str(n)] = BaggingClassifier(n_estimators=n, bootstrap=False, max_features=10)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 15.15: Example of exploring the effect of changing the number of trees in the random subspace ensemble.

Running the example first reports the mean accuracy for each configured number of decision trees.

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

compare the average outcome.

In this case, we can see that performance appears to continue to improve as the number of ensemble members is increased to 5,000.

```
>10 0.853 (0.030)
>50 0.885 (0.038)
>100 0.891 (0.034)
>500 0.894 (0.036)
>1000 0.894 (0.034)
>5000 0.896 (0.033)
```

Listing 15.16: Example output from exploring the effect of changing the number of trees in the random subspace ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees. We can see the general trend of further improvement with the number of decision trees used in the ensemble.

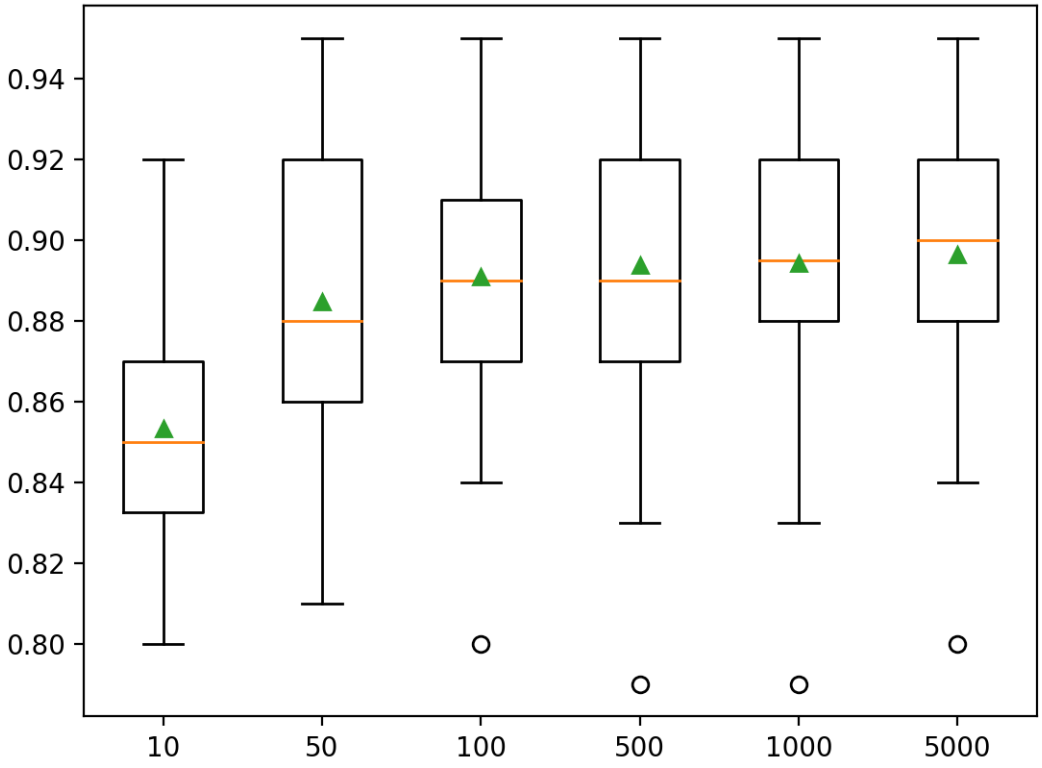

Figure 15.1: Box Plot of Random Subspace Ensemble Size vs. Classification Accuracy.

15.4.2 Explore Number of Features

The number of features selected for each random subspace controls the diversity of the ensemble. Fewer features mean more diversity, whereas more features mean less diversity. More diversity

may require more trees to reduce the variance of predictions made by the model. We can vary the diversity of the ensemble by varying the number of random features selected by setting the `max_features` argument. The example below varies the value from 1 to 20 with a fixed number of trees in the ensemble.

```
# explore random subspace ensemble number of features effect on performance
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.ensemble import BaggingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=5)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # evaluate features from 1 to 20
  for n in range(1,21):
    models[str(n)] = BaggingClassifier(n_estimators=100, bootstrap=False, max_features=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores


# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 15.17: Example of exploring the effect of changing the number of features in the random

subspace ensemble.

Running the example first reports the mean accuracy for each number of features.

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

In this case, we can see that perhaps using 8 to 11 features in the random subspaces might be appropriate on this dataset when using 100 decision trees. This might suggest increasing the number of trees to a large value first, then tuning the number of features selected in each subset.

```
>1 0.607 (0.036)
>2 0.771 (0.042)
>3 0.837 (0.036)
>4 0.858 (0.037)
>5 0.869 (0.034)
>6 0.883 (0.033)
>7 0.887 (0.038)
>8 0.894 (0.035)
>9 0.893 (0.035)
>10 0.885 (0.038)
>11 0.892 (0.034)
>12 0.883 (0.036)
>13 0.881 (0.044)
>14 0.875 (0.038)
>15 0.869 (0.041)
>16 0.861 (0.044)
>17 0.851 (0.041)
>18 0.831 (0.046)
>19 0.815 (0.046)
>20 0.801 (0.049)
```

Listing 15.18: Example output from exploring the effect of changing the number of features in the random subspace ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each number of random subset features. We can see a general trend of increasing accuracy to a point and a steady decrease in performance after 11 features.

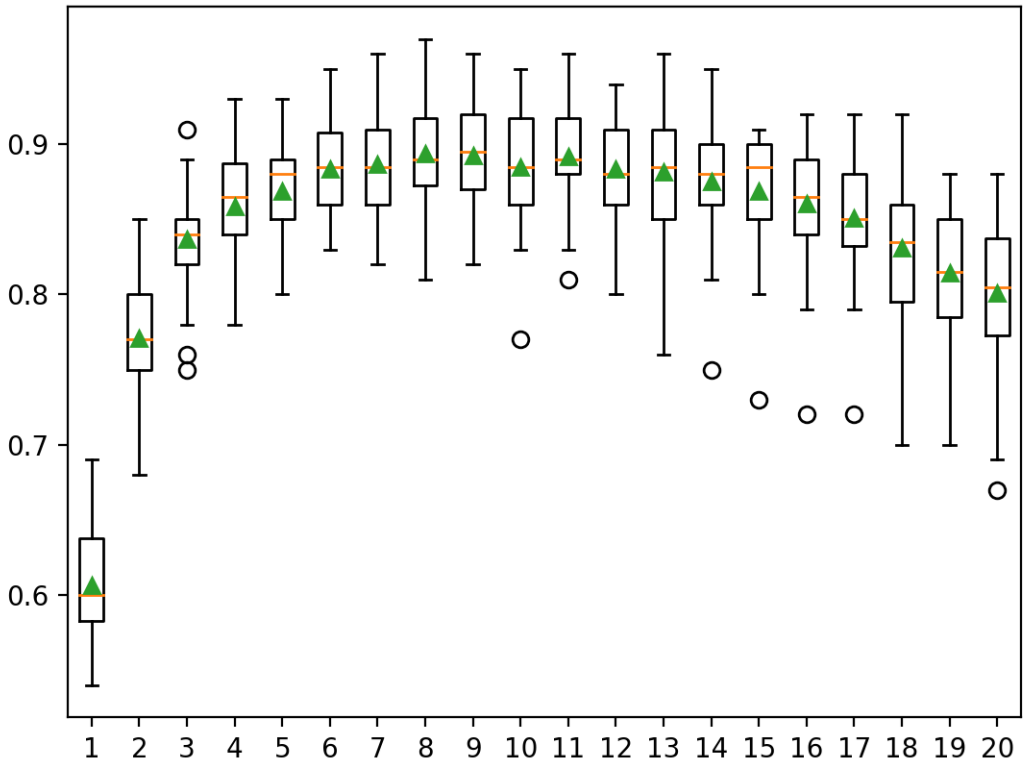

Figure 15.2: Box Plot of Random Subspace Ensemble Features vs. Classification Accuracy.

15.4.3 Explore Alternate Algorithm

Decision trees are the most common algorithm used in a random subspace ensemble. The reason for this is that they are easy to configure and work well on most problems. Other algorithms can be used to construct random subspaces and must be configured to have a modestly high variance. One example is the k-nearest neighbors algorithm where the k value can be set to a low value. The algorithm used in the ensemble is specified via the `base_estimator` argument and must be set to an instance of the algorithm and algorithm configuration to use. The example below demonstrates using a `KNeighborsClassifier` as the base algorithm used in the random subspace ensemble via the bagging class. Here, the algorithm is used with default hyperparameters where k is set to 5.

```
...
# define the model
model = BaggingClassifier(base_estimator=KNeighborsClassifier(), bootstrap=False,
    max_features=10)
```

Listing 15.19: Example of defining a random subspace ensemble with a custom base-model.

The complete example is listed below.

```
# evaluate random subspace ensemble with knn algorithm for classification
from numpy import mean
```

```
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the model
model = BaggingClassifier(base_estimator=KNeighborsClassifier(), bootstrap=False,
    max_features=10)
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 15.20: Example of exploring the effect of changing the base algorithm in the random subspace ensemble.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the random subspace ensemble with KNN and default hyperparameters achieves a classification accuracy of about 90 percent on this synthetic dataset.

```
Mean Accuracy: 0.901 (0.032)
```

Listing 15.21: Example output from exploring the effect of changing the base algorithm in the random subspace ensemble.

15.5 Further Reading

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

Papers

- *The Random Subspace Method For Constructing Decision Forests*, 1998.
  https://ieeexplore.ieee.org/abstract/document/709601/

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG

- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

APIs

- `sklearn.ensemble.BaggingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html
- `sklearn.ensemble.BaggingRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingRegressor.html

Articles

- Random subspace method, Wikipedia.
  https://en.wikipedia.org/wiki/Random_subspace_method

15.6 Summary

In this tutorial, you discovered how to develop random subspace ensembles for classification and regression. Specifically, you learned:

- Random subspace ensembles are created from decision trees fit on different samples of features (columns) in the training dataset.
- How to use the random subspace ensemble for classification and regression with scikit-learn.
- How to explore the effect of random subspace model hyperparameters on model performance.

Next

In the next section, we will take a closer look at how to develop a custom bagging ensemble using feature selection algorithms.

Chapter 16

Feature Selection Bagging Ensemble

Random subspace ensembles consist of the same model fit on different randomly selected groups of input features (columns) in the training dataset. There are many ways to choose groups of features in the training dataset, and feature selection is a popular class of data preparation techniques designed specifically for this purpose. The features selected by different configurations of the same feature selection method and different feature selection methods entirely can be used as the basis for ensemble learning. In this tutorial, you will discover how to develop feature selection subspace ensembles with Python. After completing this tutorial, you will know:

- Feature selection provides an alternative to random subspaces for selecting groups of input features.
- How to develop and evaluate ensembles composed of features selected by single feature selection techniques.
- How to develop and evaluate ensembles composed of features selected by multiple different feature selection techniques.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Feature Selection Subspace Ensemble
2. Single Feature Selection Method Ensembles
3. Combined Feature Selection Ensembles

16.2 Feature Selection Subspace Ensemble

The random subspace method or random subspace ensemble is an approach to ensemble learning that fits a model on different groups of randomly selected columns in the training dataset (introduced in Chapter 15). The difference in the choice of columns used to train each model in the ensemble results in a diversity of models and their predictions. Each model performs well, although each performs differently, making different errors.

> The training data is usually described by a set of features. Different subsets of features, or called subspaces, provide different views on the data. Therefore, individual learners trained from different subspaces are usually diverse.
>
> — Page 116, *Ensemble Methods*, 2012.

The random subspace method is often used with decision trees and the predictions made by each tree are then combined using simple statistics, such as calculating the mode class label for classification or the mean prediction for regression. Feature selection is a data preparation technique that attempts to select a subset of columns in a dataset that is most relevant to the target variable. Popular approaches involve using statistical measures, such as mutual information, and evaluating models on subsets of features and selecting the subset that results in the best performing model, called recursive feature elimination, or RFE for short.

Each feature selection method will have a different idea or informed guess about what features are most relevant to the target variable. Further, feature selection methods can be tailored to select a specific number of features from 1 to the total number of columns in the dataset, a hyperparameter that can be tuned as part of model selection. Each set of selected features may be considered as a subset of the input feature space, much like a random subspace ensemble, although chosen using a metric instead of randomly. We can use features chosen by feature selection methods as a type of ensemble model. There may be many ways that this could be implemented, but perhaps two natural approaches include:

- **One Method**: Generate a feature subspace for each number of features from 1 to the number of columns in the dataset, fit a model on each, and combine their predictions.
- **Multiple Methods**: Generate a feature subspace using multiple different feature selection methods, fit a model on each, and combine their predictions.

For lack of a better name, we can refer to this as a *Feature Selection Subspace Ensemble*. We will explore this idea in this tutorial. Let's define a test problem as the basis for this exploration and establish a baseline in performance to see if it offers a benefit over a single model. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features, five of which are redundant. The complete example is listed below.

```
# synthetic 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=5)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 16.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 16.2: Example output from creating the synthetic classification dataset.

Next, we can establish a baseline in performance. We will develop a decision tree for the dataset and evaluate it using repeated stratified k-fold cross-validation with three repeats and 10 folds. The results will be reported as the mean and standard deviation of the classification accuracy across all repeats and folds. The complete example is listed below.

```
# evaluate a decision tree on the classification dataset
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.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# define the random subspace ensemble model
model = DecisionTreeClassifier()
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 16.3: Example of evaluating a baseline model with all features on the classification dataset.

Running the example reports the mean and standard deviation classification accuracy.

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

In this case, we can see that a single decision tree model achieves a classification accuracy of approximately 79.4 percent. We can use this as a baseline in performance to see if our feature selection ensembles are able to achieve better performance.

```
Mean Accuracy: 0.794 (0.046)
```

Listing 16.4: Example output from evaluating a baseline model with all features on the classification dataset.

Next, let's explore using different feature selection methods as the basis for ensembles.

16.3 Single Feature Selection Method Ensembles

In this section, we will explore creating an ensemble from the features selected by individual feature selection methods. For a given feature selection method, we will apply it repeatedly with different numbers of selected features to create multiple feature subspaces. We will then train a model on each, in this case, a decision tree, and combine the predictions. There are many ways to combine the predictions, but to keep things simple, we will use a voting ensemble that can be configured to use hard or soft voting for classification, or averaging for regression.

We also will focus on classification and use hard voting, as the decision trees do not predict calibrated probabilities, making soft voting less appropriate.

Each model in the voting ensemble will be a `Pipeline` where the first step is a feature selection method, configured to select a specific number of features, followed by a decision tree classifier model. We will create one feature selection subspace for each number of columns in the input dataset from 1 to the number of columns. This was chosen arbitrarily for simplicity and you might want to experiment with different numbers of features in the ensemble, such as odd numbers of features, or more elaborate methods. As such, we can define a helper function named `get_ensemble()` that creates a voting ensemble via the `VotingClassifier` class with feature selection-based members for a given number of input features. We can then use this function as a template to explore using different feature selection methods. The voting ensemble for classification uses a majority vote classification of the predictions made by contributing members. We will go into a lot more detail into how voting ensembles work in Chapter 25.

```
# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # enumerate the features in the training dataset
  for i in range(1, n_features+1):
    # create the feature selection transform
    fs = ...
    # create the model
    model = DecisionTreeClassifier()
    # create the pipeline
    pipe = Pipeline([('fs', fs), ('m', model)])
    # add as a tuple to the list of models for voting
    models.append((str(i),pipe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble
```

Listing 16.5: Example of defining a function for creating a feature selection ensemble.

Given that we are working with a classification dataset, we will explore three different feature selection methods:

- ANOVA F-statistic.
- Mutual Information.
- Recursive Feature Selection.

Let's take a closer look at each.

16.3.1 ANOVA F-statistic Ensemble

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. 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.

```
# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # enumerate the features in the training dataset
  for i in range(1, n_features+1):
    # create the feature selection transform
    fs = SelectKBest(score_func=f_classif, k=i)
    # create the model
    model = DecisionTreeClassifier()
    # create the pipeline
    pipe = Pipeline([('fs', fs), ('m', model)])
    # add as a tuple to the list of models for voting
    models.append((str(i),pipe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble
```

Listing 16.6: Example of defining a function for creating an ANOVA feature selection ensemble.

Tying this together, the example below evaluates a voting ensemble composed of models fit on feature subspaces selected by the ANOVA F-statistic.

```
# example of an ensemble created from features selected with the anova f-statistic
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline

# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # enumerate the features in the training dataset
  for i in range(1, n_features+1):
    # create the feature selection transform
    fs = SelectKBest(score_func=f_classif, k=i)
    # create the model
    model = DecisionTreeClassifier()
    # create the pipeline
    pipe = Pipeline([('fs', fs), ('m', model)])
    # add as a tuple to the list of models for voting
    models.append((str(i),pipe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
```

```
  return ensemble

# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# get the ensemble model
ensemble = get_ensemble(X.shape[1])
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 16.7: Example of evaluating an ANOVA feature selection ensemble on the classification dataset.

Running the example reports the mean and standard deviation classification accuracy.

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

In this case, we can see a lift in performance over a single model that achieved an accuracy of about 79.4 percent to about 83.2 percent using an ensemble of models on features selected by the ANOVA F-statistic.

```
Mean Accuracy: 0.832 (0.043)
```

Listing 16.8: Example output from evaluating an ANOVA feature selection ensemble on the classification dataset.

Next, let's explore using mutual information.

16.3.2 Mutual Information Ensemble

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. It 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.

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).

```
# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # enumerate the features in the training dataset
  for i in range(1, n_features+1):
```

```
    # create the feature selection transform
    fs = SelectKBest(score_func=mutual_info_classif, k=i)
    # create the model
    model = DecisionTreeClassifier()
    # create the pipeline
    pipe = Pipeline([('fs', fs), ('m', model)])
    # add as a tuple to the list of models for voting
    models.append((str(i),pipe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble
```

Listing 16.9: Example of defining a function for creating an mutual information feature selection ensemble.

Tying this together, the example below evaluates a voting ensemble composed of models fit on feature subspaces selected by mutual information.

```
# example of an ensemble created from features selected with mutual information
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline

# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # enumerate the features in the training dataset
  for i in range(1, n_features+1):
    # create the feature selection transform
    fs = SelectKBest(score_func=mutual_info_classif, k=i)
    # create the model
    model = DecisionTreeClassifier()
    # create the pipeline
    pipe = Pipeline([('fs', fs), ('m', model)])
    # add as a tuple to the list of models for voting
    models.append((str(i),pipe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble

# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# get the ensemble model
ensemble = get_ensemble(X.shape[1])
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
```

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

Listing 16.10: Example of evaluating an Mutual Information feature selection ensemble on the classification dataset.

Running the example reports the mean and standard deviation classification accuracy.

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

In this case, we can see a lift in performance over using a single model, although slightly less than feature subspace selected with the Mutual Information achieving a mean accuracy of about 82.7 percent.

```
Mean Accuracy: 0.827 (0.048)
```

Listing 16.11: Example output from evaluating an Mutual Information feature selection ensemble on the classification dataset.

Next, let's explore subspaces selected using RFE.

16.3.3 Recursive Feature Selection Ensemble

Recursive Feature Elimination, or RFE for short, works by searching for a subset of features by starting with all features in the training dataset and successfully removing features until the desired number remains. This is achieved by fitting the given machine learning algorithm used in the core of the model, ranking features by importance, discarding the least important features, and re-fitting the model. This process is repeated until a specified number of features remains. The RFE method is available via the `RFE` class in scikit-learn and can be used for feature selection directly. No need to combine it with the `SelectKBest` class.

```
# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # enumerate the features in the training dataset
  for i in range(1, n_features+1):
    # create the feature selection transform
    fs = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=i)
    # create the model
    model = DecisionTreeClassifier()
    # create the pipeline
    pipe = Pipeline([('fs', fs), ('m', model)])
    # add as a tuple to the list of models for voting
    models.append((str(i),pipe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble
```

Listing 16.12: Example of defining a function for creating an RFE feature selection ensemble.

Tying this together, the example below evaluates a voting ensemble composed of models fit on feature subspaces selected by RFE.

```
# example of an ensemble created from features selected with RFE
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline

# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # enumerate the features in the training dataset
  for i in range(1, n_features+1):
    # create the feature selection transform
    fs = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=i)
    # create the model
    model = DecisionTreeClassifier()
    # create the pipeline
    pipe = Pipeline([('fs', fs), ('m', model)])
    # add as a tuple to the list of models for voting
    models.append((str(i),pipe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble

# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=5)
# get the ensemble model
ensemble = get_ensemble(X.shape[1])
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 16.13: Example of evaluating an RFE feature selection ensemble on the classification dataset.

Running the example reports the mean and standard deviation classification accuracy.

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

In this case, we can see that the mean accuracy is similar to that seen with mutual information feature selection, with a score of about 82.3 percent.

```
Mean Accuracy: 0.823 (0.045)
```

Listing 16.14: Example output from evaluating an RFE feature selection ensemble on the classification dataset.

This is a good start, and it might be interesting to see if better results can be achieved using ensembles composed of fewer members, e.g. every second, third, or fifth number of selected features. Next, let's see if we can improve results by combining models fit on feature subspaces selected by different feature selection methods.

16.4 Combined Feature Selection Ensembles

In the previous section, we saw that we can get a lift in performance over a single model by using a single feature selection method as the basis of an ensemble prediction for a dataset. We would expect the predictions between many of the members of the ensemble to be correlated. This could be addressed by using different numbers of selected input features as the basis for the ensemble rather than a contiguous number of features from 1 to the number of columns. An alternative approach to introducing diversity is to select feature subspaces using different feature selection methods. We will explore two versions of this approach. With the first, we will select the same number of features from each method, and with the second, we will select a contiguous number of features from 1 to the number of columns for multiple methods.

16.4.1 Ensemble With Fixed Number of Features

In this section, we will make our first attempt at devising an ensemble using features selected by multiple feature selection techniques. We will select an arbitrary number of features from the dataset, then use each of the three feature selection methods to select a feature subspace, fit a model of each, and use them as the basis for a voting ensemble. The `get_ensemble()` function below implements this, taking the specified number of features to select with each method as an argument. The hope is that the features selected by each method are sufficiently different and sufficiently skillful to result in an effective ensemble.

```
# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # anova member
  fs = SelectKBest(score_func=f_classif, k=n_features)
  anova = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('anova', anova))
  # mutual information member
  fs = SelectKBest(score_func=mutual_info_classif, k=n_features)
  mutinfo = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('mutinfo', mutinfo))
  # rfe member
  fs = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=n_features)
  rfe = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('rfe', rfe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
```

```
  return ensemble
```

Listing 16.15: Example of defining a function for creating a combined feature selection ensemble with a fixed number of features.

Tying this together, the example below evaluates an ensemble of a fixed number of features selected using different feature selection methods.

```
# ensemble of a fixed number of features selected by different feature selection methods
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFE
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import f_classif
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline

# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models = list()
  # anova member
  fs = SelectKBest(score_func=f_classif, k=n_features)
  anova = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('anova', anova))
  # mutual information member
  fs = SelectKBest(score_func=mutual_info_classif, k=n_features)
  mutinfo = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('mutinfo', mutinfo))
  # rfe member
  fs = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=n_features)
  rfe = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('rfe', rfe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble

# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1)
# get the ensemble model
ensemble = get_ensemble(15)
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 16.16: Example of evaluating an combined feature selection ensemble with a fixed number of features on the classification dataset.

Running the example reports the mean and standard deviation classification accuracy.

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

In this case, we can see a modest lift in performance over the techniques considered in the previous section, resulting in a mean classification accuracy of about 83.9 percent.

```
Mean Accuracy: 0.839 (0.044)
```

Listing 16.17: Example output from evaluating an combined feature selection ensemble with a fixed number of features on the classification dataset.

A more fair comparison might be to compare this result to each individual model that comprises the ensemble. The updated example performs exactly this comparison.

```
# comparison of ensemble of a fixed number of features to standalone models
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFE
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import f_classif
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# get a voting ensemble of models
def get_ensemble(n_features):
  # define the base models
  models, names = list(), list()
  # anova member
  fs = SelectKBest(score_func=f_classif, k=n_features)
  anova = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('anova', anova))
  names.append('anova')
  # mutual information member
  fs = SelectKBest(score_func=mutual_info_classif, k=n_features)
  mutinfo = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('mutinfo', mutinfo))
  names.append('mutinfo')
  # rfe member
  fs = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=n_features)
  rfe = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
  models.append(('rfe', rfe))
  names.append('rfe')
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  names.append('ensemble')
  return names, [anova, mutinfo, rfe, ensemble]
```

```
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1)
# get the ensemble model
names, models = get_ensemble(15)
# evaluate each model
results = list()
for model,name in zip(models,names):
  # define the evaluation method
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model on the dataset
  n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store the results
  results.append(n_scores)
  # report performance
  print('>%s: %.3f (%.3f)' % (name, mean(n_scores), std(n_scores)))
# plot the results for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 16.18: Example of comparing models with feature selection to combined feature selection ensemble with a fixed number of features.

Running the example reports the mean performance of each single model fit on the selected features and ends with the performance of the ensemble that combines all three models.

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

In this case, the results suggest that the ensemble of the models fit on the selected features performs better than any single model in the ensemble, as we might hope.

```
>anova: 0.811 (0.048)
>mutinfo: 0.807 (0.041)
>rfe: 0.825 (0.043)
>ensemble: 0.837 (0.040)
```

Listing 16.19: Example output from comparing models with feature selection to combined feature selection ensemble with a fixed number of features.

A figure is created to show box and whisker plots for each set of results, allowing the distribution accuracy scores to be compared directly. We can see that the distribution for the ensemble both skews higher and has a larger median classification accuracy (orange line), visually confirming the finding.

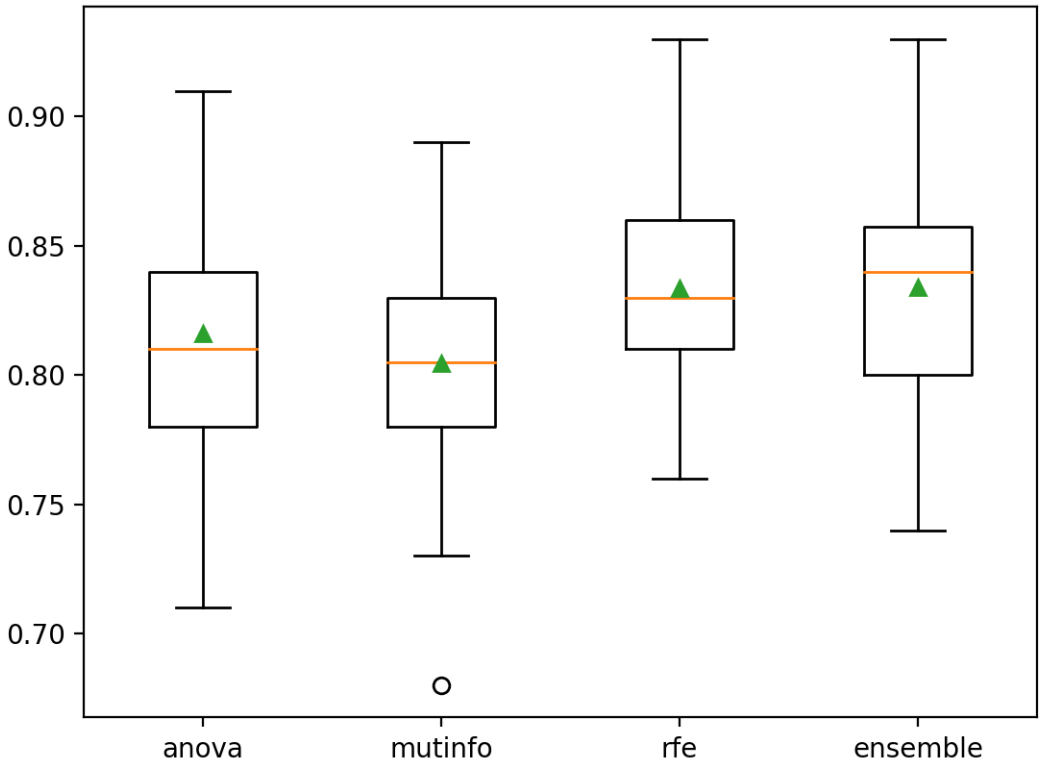

Figure 16.1: Box and Whisker Plots of Accuracy of Singles Model Fit On Selected Features vs. Ensemble.

Next, let's explore adding multiple members for each feature selection method.

16.4.2 Ensemble With Contiguous Number of Features

We can combine the experiments from the previous section. Specifically, we can select multiple feature subspaces using each feature selection method, fit a model on each, and add all of the models to a single ensemble. In this case, we will select subspaces as we did in the previous section from 1 to the number of columns in the dataset, although in this case, repeat the process with each feature selection method.

```
# get a voting ensemble of models
def get_ensemble(n_features_start, n_features_end):
  # define the base models
  models = list()
  for i in range(n_features_start, n_features_end+1):
    # anova
    fs = SelectKBest(score_func=f_classif, k=i)
    anova = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
    models.append(('anova'+str(i), anova))
    # mutual information
    fs = SelectKBest(score_func=mutual_info_classif, k=i)
```

```
    mutinfo = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
    models.append(('mutinfo'+str(i), mutinfo))
    # rfe
    fs = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=i)
    rfe = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
    models.append(('rfe'+str(i), rfe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble
```

Listing 16.20: Example of defining a function for creating a combined feature selection ensemble with contiguous numbers of features.

The hope is that the diversity of the selected features across the feature selection methods results in a further lift in ensemble performance. Tying this together, the complete example is listed below.

```
# ensemble of many subsets of features selected by multiple feature selection methods
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.feature_selection import RFE
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
from sklearn.feature_selection import f_classif
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline

# get a voting ensemble of models
def get_ensemble(n_features_start, n_features_end):
  # define the base models
  models = list()
  for i in range(n_features_start, n_features_end+1):
    # anova member
    fs = SelectKBest(score_func=f_classif, k=i)
    anova = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
    models.append(('anova'+str(i), anova))
    # mutual information member
    fs = SelectKBest(score_func=mutual_info_classif, k=i)
    mutinfo = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
    models.append(('mutinfo'+str(i), mutinfo))
    # rfe member
    fs = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=i)
    rfe = Pipeline([('fs', fs), ('m', DecisionTreeClassifier())])
    models.append(('rfe'+str(i), rfe))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble

# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1)
# get the ensemble model
```

```
ensemble = get_ensemble(1, 20)
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 16.21: Example of evaluating an combined feature selection ensemble with a contiguous features on the classification dataset.

Running the example reports the mean and standard deviation classification accuracy of the ensemble.

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

In this case, we can see a further lift of performance as we hoped, where the combined ensemble resulted in a mean classification accuracy of about 86.0 percent.

```
Mean Accuracy: 0.860 (0.036)
```

Listing 16.22: Example output from evaluating an combined feature selection ensemble with a contiguous features on the classification dataset.

The use of feature selection for selecting subspaces of input features may provide an interesting alternative or perhaps complement to selecting random subspaces.

16.5 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

APIs

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

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

Articles

- Random subspace method, Wikipedia.
  https://en.wikipedia.org/wiki/Random_subspace_method

16.6 Summary

In this tutorial, you discovered how to develop feature selection subspace ensembles with Python. Specifically, you learned:

- Feature selection provides an alternative to random subspaces for selecting groups of input features.
- How to develop and evaluate ensembles composed of features selected by single feature selection techniques.
- How to develop and evaluate ensembles composed of features selected by multiple different feature selection techniques.

Next

In the next section, we will take a closer look at an extension to bagging and perhaps the most popular type of algorithm of this class called the random forest ensemble.

Chapter 17

Random Forest Ensemble

Random forest is an ensemble machine learning algorithm. It is perhaps the most popular and widely used machine learning algorithm given its good to excellent performance across a wide range of classification and regression predictive modeling problems. It is also easy to use given that it has few key hyperparameters and sensible heuristics for configuring these hyperparameters. In this tutorial, you will discover how to develop a random forest ensemble for classification and regression. After completing this tutorial, you will know:

- Random forest ensemble is an ensemble of decision trees and a natural extension of bagging.
- How to use the random forest ensemble for classification and regression with scikit-learn.
- How to explore the effect of random forest model hyperparameters on model performance.

Let's get started.

17.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Random Forest Algorithm
2. Evaluate Random Forest Ensembles
3. Random Forest Hyperparameters
4. Common Questions

17.2 Random Forest Algorithm

Random forest is an ensemble of decision tree algorithms. It is an extension of bootstrap aggregation (bagging) of decision trees and can be used for classification and regression problems. In bagging, a number of decision trees are created where each tree is created from a different bootstrap sample of the training dataset. A bootstrap sample is a sample of the training dataset where an instance (row) may be selected more than once. This is referred to as *sampling with replacement* (of instances).

Bagging is an effective ensemble algorithm as each decision tree is fit on a slightly different training dataset, and in turn, has a slightly different performance. Unlike normal decision tree models, such as classification and regression trees (CART), trees used in the ensemble are unpruned, making them slightly overfit to the training dataset. This is desirable as it helps to make each tree more different and have less correlated predictions or prediction errors. Predictions from the trees are averaged across all decision trees resulting in better performance than any single tree in the model.

> Each model in the ensemble is then used to generate a prediction for a new sample and these m predictions are averaged to give the forest's prediction
>
> — Page 199, *Applied Predictive Modeling*, 2013.

A prediction on a regression problem is the average of the prediction across the trees in the ensemble. A prediction on a classification problem is the majority vote for the class label across the trees in the ensemble.

- **Regression**: Prediction is the average prediction across the decision trees.
- **Classification**: Prediction is the majority vote class label predicted across the decision trees.

Random forest involves constructing a large number of decision trees from bootstrap samples from the training dataset, like bagging.

> As with bagging, each tree in the forest casts a vote for the classification of a new sample, and the proportion of votes in each class across the ensemble is the predicted probability vector.
>
> — Page 387, *Applied Predictive Modeling*, 2013.

Unlike bagging, random forest also involves selecting a subset of input features (columns or variables) at each split point in the construction of trees. Typically, constructing a decision tree involves evaluating the value for each input variable in the data in order to select a split point. By reducing the features to a random subset that may be considered at each split point, it forces each decision tree in the ensemble to be more different.

> Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees. [...] But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors.
>
> — Page 320, *An Introduction to Statistical Learning with Applications in R*, 2014.

The effect is that the predictions, and in turn, prediction errors, made by each tree in the ensemble are more different or less correlated. When the predictions from these less correlated trees are averaged to make a prediction, it often results in better performance than bagged decision trees. Perhaps the most important hyperparameter to tune for the random forest is the number of random features to consider at each split point.

> Random forests' tuning parameter is the number of randomly selected predictors, k, to choose from at each split, and is commonly referred to as *mtry*. In the regression context, Breiman recommends setting *mtry* to be one-third of the number of predictors.
>
> — Page 199, *Applied Predictive Modeling*, 2013.

A good heuristic for regression is to set this hyperparameter to $\frac{1}{3}$ the number of input features.

$$\text{num\_features\_for\_split} = \frac{\text{total\_input\_features}}{3} \tag{17.1}$$

> For classification problems, Breiman recommends setting *mtry* to the square root of the number of predictors.
>
> — Page 387, *Applied Predictive Modeling*, 2013.

A good heuristic for classification is to set this hyperparameter to the square root of the number of input features.

$$\text{num\_features\_for\_split} = \sqrt{\text{total\_input\_features}} \tag{17.2}$$

Another important hyperparameter to tune is the depth of the decision trees. Deeper trees are often more overfit to the training data, but also less correlated, which in turn may improve the performance of the ensemble. Depths from 1 to 10 levels may be effective. Finally, the number of decision trees in the ensemble can be set. Often, this is increased until no further improvement is seen.

17.3 Evaluate Random Forest Ensembles

The scikit-learn Python machine learning library provides an implementation of Random Forest for machine learning via the `RandomForestRegressor` and `RandomForestClassifier` classes. Both models operate the same way and take the same arguments that influence how the decision trees are created. Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model. When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions. Let's take a look at how to develop a Random Forest ensemble for both classification and regression tasks.

17.3.1 Random Forest for Classification

In this section, we will look at using Random Forest for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=3)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 17.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 17.2: Example output from creating the synthetic classification dataset.

Next, we can evaluate a random forest algorithm on this dataset. We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate random forest algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import RandomForestClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=3)
# define the model
model = RandomForestClassifier()
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 17.3: Example of evaluating the random forest ensemble on the synthetic classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the random forest ensemble with default hyperparameters achieves a classification accuracy of about 90.5 percent.

```
Mean Accuracy: 0.905 (0.025)
```

Listing 17.4: Example output from evaluating the random forest ensemble the synthetic classification dataset.

We can also use the random forest model as a final model and make predictions for classification. First, the random forest ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make predictions using random forest for classification
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=3)
# define the model
model = RandomForestClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [-8.52381793, 5.24451077, -12.14967704, -2.92949242, 0.99314133, 0.67326595,
    -0.38657932, 1.27955683, -0.60712621, 3.20807316, 0.60504151, -1.38706415, 8.92444588,
    -7.43027595, -2.33653219, 1.10358169, 0.21547782, 1.05057966, 0.6975331, 0.26076035]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 17.5: Example of making a prediction with the random forest ensemble on the synthetic classification dataset.

Running the example fits the random forest ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 0
```

Listing 17.6: Example output from making a prediction with the random forest ensemble on the synthetic classification dataset.

Now that we are familiar with using random forest for classification, let's look at the API for regression.

17.3.2 Random Forest for Regression

In this section, we will look at using random forests for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=2)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 17.7: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 17.8: Example output from creating the synthetic regression dataset.

Next, we can evaluate a random forest algorithm on this dataset. As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate random forest ensemble for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.ensemble import RandomForestRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=2)
# define the model
model = RandomForestRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 17.9: Example of evaluating the random forest ensemble on the synthetic regression dataset.

Running the example reports the mean and standard deviation MAE of the model.

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

In this case, we can see the random forest ensemble with default hyperparameters achieves a MAE of about 90.

```
MAE: -90.149 (7.924)
```

Listing 17.10: Example output from evaluating the random forest ensemble the synthetic regression dataset.

We can also use the random forest model as a final model and make predictions for regression. First, the random forest ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# random forest for making predictions for regression
from sklearn.datasets import make_regression
from sklearn.ensemble import RandomForestRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=2)
# define the model
model = RandomForestRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [-0.89483109, -1.0670149, -0.25448694, -0.53850126, 0.21082105, 1.37435592,
    0.71203659, 0.73093031, -1.25878104, -2.01656886, 0.51906798, 0.62767387, 0.96250155,
    1.31410617, -1.25527295, -0.85079036, 0.24129757, -0.17571721, -1.11454339, 0.36268268]
yhat = model.predict([row])
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Listing 17.11: Example of making a prediction with the random forest ensemble on the synthetic regression dataset.

Running the example fits the random forest ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: -173
```

Listing 17.12: Example output from making a prediction with the random forest ensemble on the synthetic regression dataset.

Now that we are familiar with using the scikit-learn API to evaluate and use random forest ensembles, let's look at configuring the model.

17.4 Random Forest Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the random forest ensemble and their effect on model performance.

17.4.1 Explore Number of Samples

Each decision tree in the ensemble is fit on a bootstrap sample drawn from the training dataset. This can be turned off by setting the `bootstrap` argument to `False`, if you desire. In that case, the whole training dataset will be used to train each decision tree. The `max_samples` argument can be set to a float between 0 and 1 to control the percentage of the size of the training dataset to make the bootstrap sample used to train each decision tree.

For example, if the training dataset has 100 rows, the `max_samples` argument could be set to 0.5 and each decision tree will be fit on a bootstrap sample with (100×0.5) or 50 rows of data. A smaller sample size will make trees more different, and a larger sample size will make the trees more similar. Setting `max_samples` to `None` will make the sample size the same size as the training dataset and this is the default. The example below demonstrates the effect of different bootstrap sample sizes from 10 percent to 100 percent on the random forest algorithm.

```
# explore random forest bootstrap sample size on performance
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=3)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore ratios from 10% to 100% in 10% increments
  for i in arange(0.1, 1.1, 0.1):
    key = '%.1f' % i
    # set max_samples=None to use 100%
    if i == 1.0:
      i = None
    models[key] = RandomForestClassifier(max_samples=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 17.13: Example of exploring the effect of the number of samples on the random forest

ensemble.

Running the example first reports the mean accuracy for each dataset size.

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

In this case, the results suggest that using a bootstrap sample size that is equal to the size of the training dataset achieves the best results on this dataset. This is the default and it should probably be used in most cases.

```
>0.1 0.857 (0.031)
>0.2 0.875 (0.028)
>0.3 0.886 (0.025)
>0.4 0.896 (0.027)
>0.5 0.897 (0.025)
>0.6 0.894 (0.028)
>0.7 0.897 (0.028)
>0.8 0.897 (0.026)
>0.9 0.906 (0.023)
>1.0 0.906 (0.021)
```

Listing 17.14: Example output from exploring the effect of the number of samples on the random forest ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each bootstrap sample size. In this case, we can see a general trend that the larger the sample, the better the performance of the model. You might like to extend this example and see what happens if the bootstrap sample size is larger or even much larger than the training dataset (e.g. you can set an integer value as the number of samples instead of a float percentage of the training dataset size).

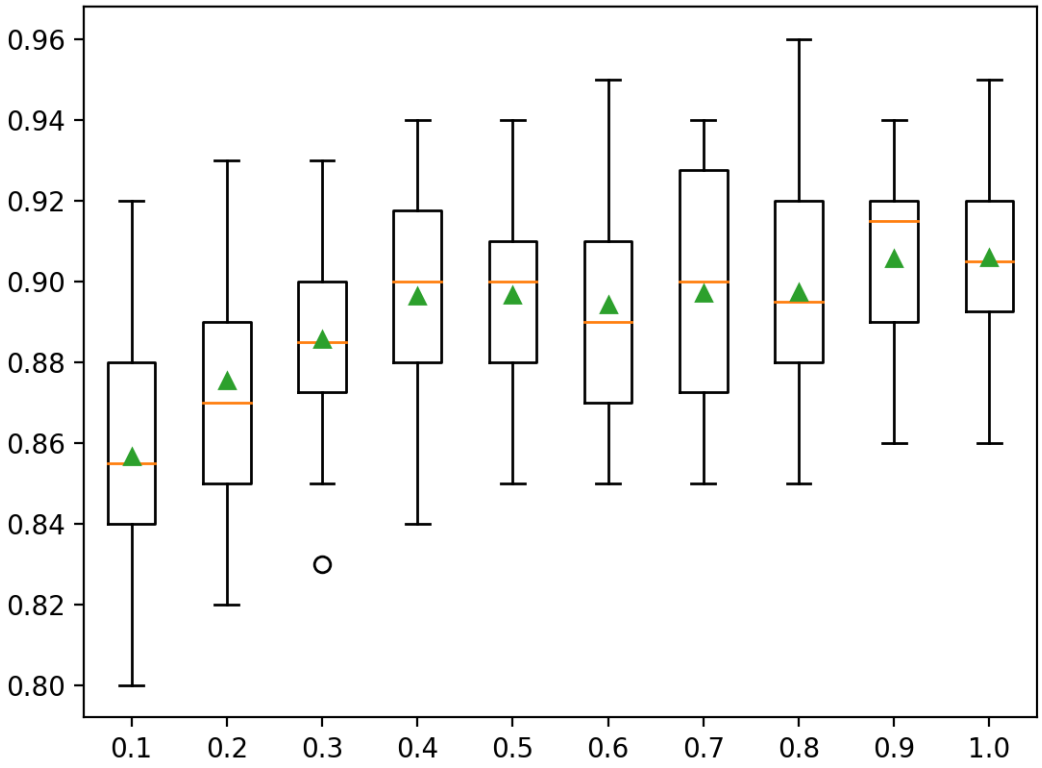

Figure 17.1: Box Plot of Random Forest Bootstrap Sample Size vs. Classification Accuracy.

17.4.2 Explore Number of Features

The number of features that is randomly sampled for each split point is perhaps the most important feature to configure for random forest. It is set via the `max_features` argument and defaults to the square root of the number of input features. In this case, for our synthetic dataset, this would be $\sqrt{20}$ or about four features. The example below explores the effect of the number of features randomly selected at each split point on model accuracy. We will try values from 1 to 7 and would expect a small value, around four, to perform well based on the heuristic.

```
# explore random forest number of features effect on performance
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.ensemble import RandomForestClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=3)
```

```
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore number of features from 1 to 7
  for i in range(1,8):
    models[str(i)] = RandomForestClassifier(max_features=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 17.15: Example of exploring the effect of the number of features on the random forest ensemble.

Running the example first reports the mean accuracy for each feature set size.

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

In this case, the results suggest that a value between three and five would be appropriate, confirming the sensible default of four on this dataset. A value of five might even be better given the smaller standard deviation in classification accuracy as compared to a value of three or four.

```
>1 0.897 (0.023)
>2 0.900 (0.028)
>3 0.903 (0.027)
>4 0.903 (0.022)
>5 0.903 (0.019)
```

```
>6 0.898 (0.025)
>7 0.900 (0.024)
```

Listing 17.16: Example output from exploring the effect of the number of features on the random forest ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each feature set size. We can see a trend in performance rising and peaking with values between three and five and falling again as larger feature set sizes are considered.

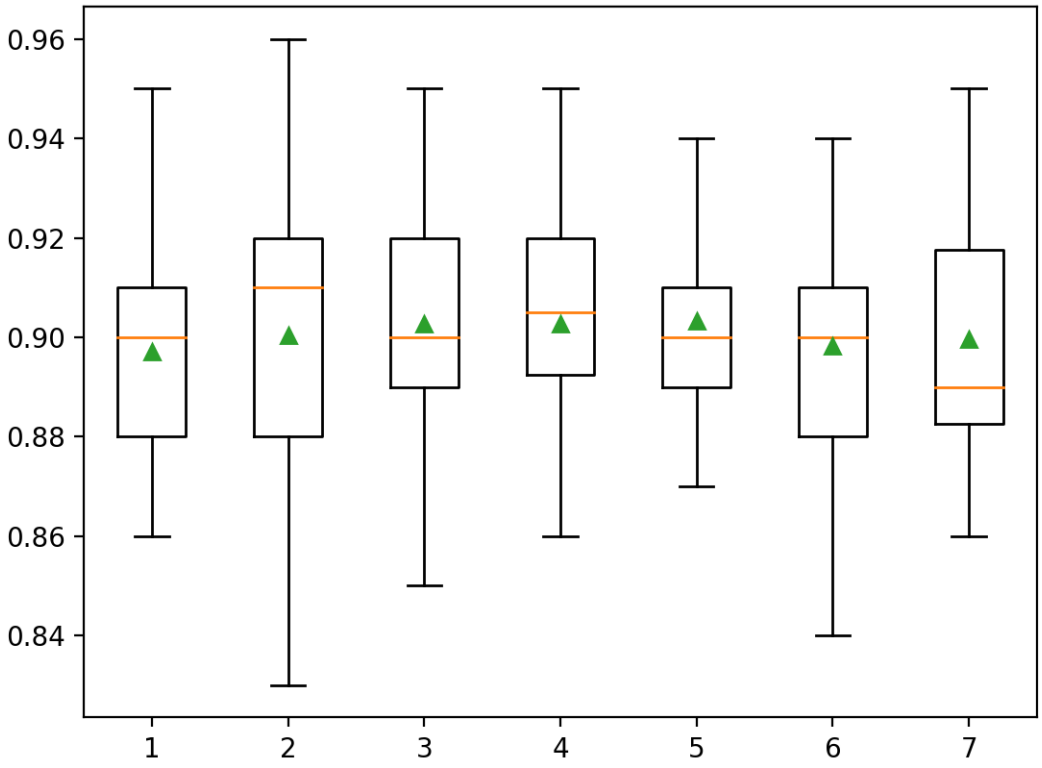

Figure 17.2: Box Plot of Random Forest Feature Set Size vs. Classification Accuracy.

17.4.3 Explore Number of Trees

The number of trees is another key hyperparameter to configure for the random forest. Typically, the number of trees is increased until the model performance stabilizes. Intuition might suggest that more trees will lead to overfitting, although this is not the case. Both bagging and random forest algorithms appear to be somewhat immune to overfitting the training dataset given the stochastic nature of the learning algorithm. The number of trees can be set via the `n_estimators` argument and defaults to 100. The example below explores the effect of the number of trees with values between 10 to 1,000.

```
# explore random forest number of trees effect on performance
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.ensemble import RandomForestClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=3)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define number of trees to consider
  n_trees = [10, 50, 100, 500, 1000]
  for n in n_trees:
    models[str(n)] = RandomForestClassifier(n_estimators=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 17.17: Example of exploring the effect of the number of trees on the random forest ensemble.

Running the example first reports the mean accuracy for each configured number of trees.

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

In this case, we can see that performance rises and stays flat after about 100 trees. Mean accuracy scores fluctuate across 100, 500, and 1,000 trees and this may be statistical noise.

```
>10 0.870 (0.036)
>50 0.900 (0.028)
>100 0.910 (0.024)
>500 0.904 (0.024)
>1000 0.906 (0.023)
```

Listing 17.18: Example output from exploring the effect of the number of trees on the random forest ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees.

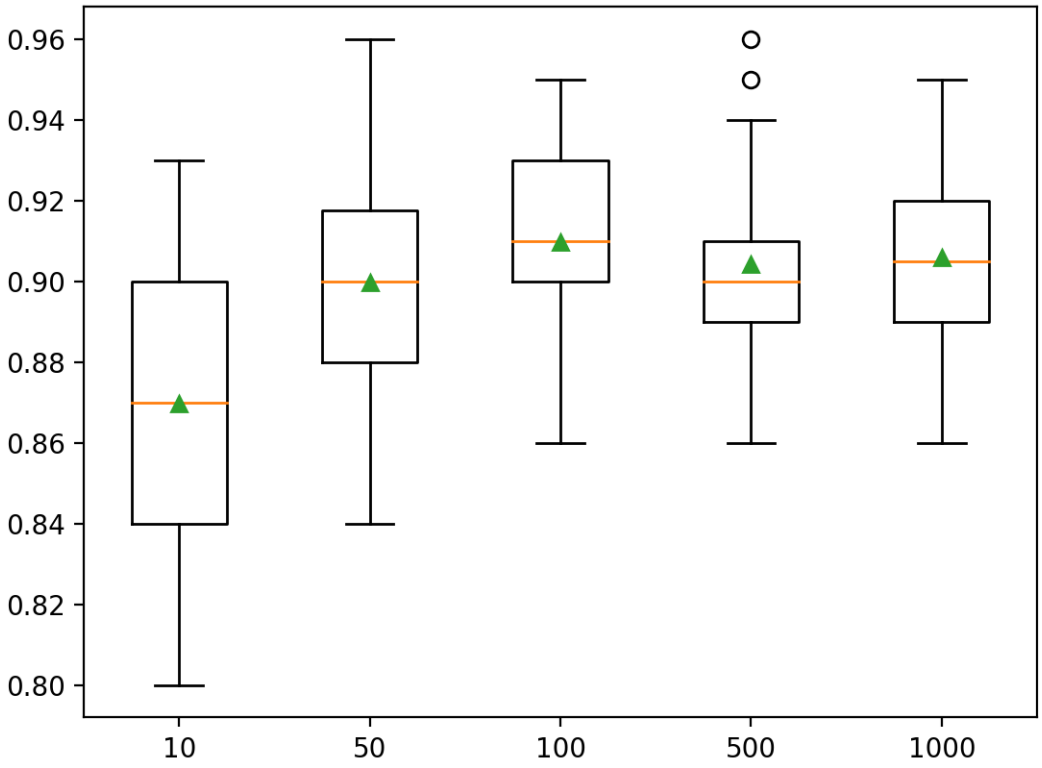

Figure 17.3: Box Plot of Random Forest Ensemble Size vs. Classification Accuracy.

17.4.4 Explore Tree Depth

A final interesting hyperparameter is the maximum depth of decision trees used in the ensemble. By default, trees are constructed to an arbitrary depth and are not pruned. This is a sensible default, although we can also explore fitting trees with different fixed depths. The maximum tree depth can be specified via the `max_depth` argument and is set to None (no maximum depth) by default. The example below explores the effect of random forest maximum tree depth on model performance.

```
# explore random forest tree depth effect on performance
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.ensemble import RandomForestClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=3)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # consider tree depths from 1 to 7 and None=full
  depths = [i for i in range(1,8)] + [None]
  for n in depths:
    models[str(n)] = RandomForestClassifier(max_depth=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 17.19: Example of exploring the effect of the tree depth on the random forest ensemble.

Running the example first reports the mean accuracy for each configured maximum tree depth.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation

procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that larger depth results in better model performance, with the default of no maximum depth achieving the best performance on this dataset.

```
>1 0.771 (0.040)
>2 0.807 (0.037)
>3 0.834 (0.034)
>4 0.857 (0.030)
>5 0.872 (0.025)
>6 0.887 (0.024)
>7 0.890 (0.025)
>None 0.903 (0.027)
```

Listing 17.20: Example output from exploring the effect of the tree depth on the random forest ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured maximum tree depth. In this case, we can see a trend of improved performance with increase in tree depth, supporting the default of no maximum depth.

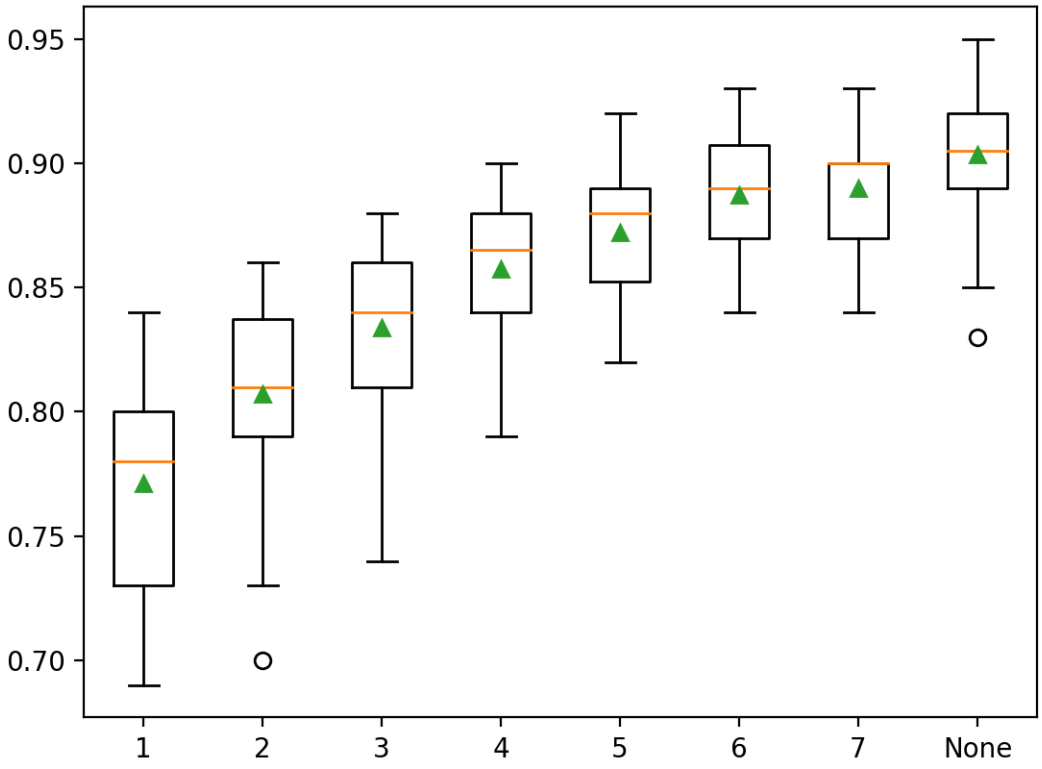

Figure 17.4: Box Plot of Random Forest Maximum Tree Depth vs. Classification Accuracy.

17.5 Common Questions

In this section we will take a closer look at some common sticking points you may have with the random forest ensemble procedure.

Q. What algorithm should be used in the ensemble?

Random forest is designed to be an ensemble of decision tree algorithms.

Q. How many ensemble members should be used?

The number of trees should be increased until no further improvement in performance is seen on your dataset.

> As a starting point, we suggest using at least 1,000 trees. If the cross-validation performance profiles are still improving at 1,000 trees, then incorporate more trees until performance levels off.
>
> — Page 200, *Applied Predictive Modeling*, 2013.

Q. Won't the ensemble overfit with too many trees?

No. Random forest ensembles (do not) are very unlikely to overfit in general.

> Another claim is that random forests "cannot overfit" the data. It is certainly true that increasing [the number of trees] does not cause the random forest sequence to overfit ...
>
> — Page 596, *The Elements of Statistical Learning*, 2016.

Q. How large should the bootstrap sample be?

It is good practice to make the bootstrap sample as large as the original dataset size. That is 100% the size or an equal number of rows as the original dataset.

Q. How many features should be chosen at each split point?

The best practice is to test a suite of different values and discover what works best for your dataset. As a heuristic, you can use:

- **Classification**: Square root of the number of input features.
- **Regression**: One third of the number of input features.

Q. What problems are well suited to random forest?

Random forest is known to work well or even best on a wide range of classification and regression problems. Try it and see.

> The authors make grand claims about the success of random forests: "most accurate", "most interpretable", and the like. In our experience random forests do remarkably well, with very little tuning required.
>
> — Page 590, *The Elements of Statistical Learning*, 2016.

17.6 Further Reading

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

Papers

- *Random Forests*, 2001.
  https://link.springer.com/article/10.1023/A:1010933404324

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5
- *Applied Predictive Modeling*, 2013.
  https://amzn.to/2O3BuOa
- *An Introduction to Statistical Learning with Applications in R*, 2014.
  https://amzn.to/37xa7DT
- *The Elements of Statistical Learning*, 2016..
  https://amzn.to/3ahyI1O

APIs

- `sklearn.ensemble.RandomForestRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
- `sklearn.ensemble.RandomForestClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Articles

- Random Forest, Wikipedia.
  https://en.wikipedia.org/wiki/Random_forest

17.7 Summary

In this tutorial, you discovered how to develop random forest ensembles for classification and regression. Specifically, you learned:

- Random forest ensemble is an ensemble of decision trees and a natural extension of bagging.
- How to use the random forest ensemble for classification and regression with scikit-learn.
- How to explore the effect of random forest model hyperparameters on model performance.

Next

In the next section, we will take a closer look at another extension to bagging and sibling to random forest called extra trees ensembles.

Chapter 18

Extra Trees Ensemble

Extra Trees is an ensemble machine learning algorithm that combines the predictions from many decision trees. It is related to the widely used random forest algorithm. It can often achieve as-good or better performance than the random forest algorithm, although it uses a simpler algorithm to construct the decision trees used as members of the ensemble. It is also easy to use given that it has few key hyperparameters and sensible heuristics for configuring these hyperparameters. In this tutorial, you will discover how to develop Extra Trees ensembles for classification and regression. After completing this tutorial, you will know:

- Extra Trees ensemble is an ensemble of decision trees and is related to bagging and random forest.
- How to use the Extra Trees ensemble for classification and regression with scikit-learn.
- How to explore the effect of Extra Trees model hyperparameters on model performance.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Extra Trees Algorithm
2. Evaluate Extra Trees Ensembles
3. Extra Trees Hyperparameters

18.2 Extra Trees Algorithm

Extremely Randomized Trees, or Extra Trees for short, is an ensemble machine learning algorithm. Specifically, it is an ensemble of decision trees and is related to other ensembles of decision trees algorithms such as bootstrap aggregation (bagging) and random forest. The Extra Trees algorithm works by creating a large number of unpruned decision trees from the training dataset. Predictions are made by averaging the prediction of the decision trees in the case of regression or using majority voting in the case of classification.

- **Regression**: Predictions made by averaging predictions from decision trees.
- **Classification**: Predictions made by majority voting from decision trees.

> The predictions of the trees are aggregated to yield the final prediction, by majority vote in classification problems and arithmetic average in regression problems.
>
> — *Extremely Randomized Trees*, 2006.

Unlike bagging and random forest that develop each decision tree from a bootstrap sample of the training dataset, the Extra Trees algorithm fits each decision tree on the whole training dataset. Like random forest, the Extra Trees algorithm will randomly sample the features at each split point of a decision tree. Unlike random forest, which uses a greedy algorithm to select an optimal split point, the Extra Trees algorithm selects a split point at random.

> The Extra-Trees algorithm builds an ensemble of unpruned decision or regression trees according to the classical top-down procedure. Its two main differences with other tree-based ensemble methods are that it splits nodes by choosing cut-points fully at random and that it uses the whole learning sample (rather than a bootstrap replica) to grow the trees.
>
> — *Extremely Randomized Trees*, 2006.

As such, there are three main hyperparameters to tune in the algorithm; they are the number of decision trees in the ensemble, the number of input features to randomly select and consider for each split point, and the minimum number of samples required in a node to create a new split point.

> It has two parameters: K, the number of attributes randomly selected at each node and *nmin*, the minimum sample size for splitting a node. [...] we denote by M the number of trees of this ensemble.
>
> — *Extremely Randomized Trees*, 2006.

The random selection of split points makes the decision trees in the ensemble less correlated, although this increases the variance of the algorithm. This increase in variance can be countered by increasing the number of trees used in the ensemble.

> The parameters K, *nmin* and M have different effects: K determines the strength of the attribute selection process, *nmin* the strength of averaging output noise, and M the strength of the variance reduction of the ensemble model aggregation.
>
> — *Extremely Randomized Trees*, 2006.

18.3 Evaluate Extra Trees Ensembles

The scikit-learn Python machine learning library provides an implementation of Extra Trees for machine learning via the `ExtraTreesRegressor` and `ExtraTreesClassifier` classes. Both models operate the same way and take the same arguments that influence how the decision trees are created. Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model. When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions. Let's take a look at how to develop an Extra Trees ensemble for both classification and regression.

18.3.1 Extra Trees for Classification

In this section, we will look at using Extra Trees for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=4)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 18.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 18.2: Example output from creating the synthetic classification dataset.

Next, we can evaluate an Extra Trees algorithm on this dataset. We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate extra trees algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import ExtraTreesClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=4)
# define the model
model = ExtraTreesClassifier()
# define the evaluation method
```

```
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 18.3: Example of evaluating the extra trees ensemble on the synthetic classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Extra Trees ensemble with default hyperparameters achieves a classification accuracy of about 91 percent on this synthetic dataset.

```
Mean Accuracy: 0.910 (0.027)
```

Listing 18.4: Example output from evaluating the extra trees ensemble the synthetic classification dataset.

We can also use the Extra Trees model as a final model and make predictions for classification. First, the Extra Trees ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make predictions using extra trees for classification
from sklearn.datasets import make_classification
from sklearn.ensemble import ExtraTreesClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=4)
# define the model
model = ExtraTreesClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [-3.52169364, 4.00560592, 2.94756812, -0.09755101, -0.98835896, 1.81021933,
    -0.32657994, 1.08451928, 4.98150546, -2.53855736, 3.43500614, 1.64660497, -4.1557091,
    -1.55301045, -0.30690987, -1.47665577, 6.818756, 0.5132918, 4.3598337, -4.31785495]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 18.5: Example of making a prediction with the extra trees ensemble on the synthetic classification dataset.

Running the example fits the Extra Trees ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 0
```

Listing 18.6: Example output from making a prediction with the extra trees ensemble on the synthetic classification dataset.

Now that we are familiar with using Extra Trees for classification, let's look at the API for regression.

18.3.2 Extra Trees for Regression

In this section, we will look at using Extra Trees for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=3)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 18.7: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 18.8: Example output from creating the synthetic regression dataset.

Next, we can evaluate an Extra Trees algorithm on this dataset. As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate extra trees ensemble for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.ensemble import ExtraTreesRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=3)
# define the model
model = ExtraTreesRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 18.9: Example of evaluating the extra trees ensemble on the synthetic regression dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Extra Trees ensemble with default hyperparameters achieves a MAE of about 70.

```
MAE: -69.561 (5.616)
```

Listing 18.10: Example output from evaluating the extra trees ensemble the synthetic regression dataset.

We can also use the Extra Trees model as a final model and make predictions for regression. First, the Extra Trees ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# extra trees for making predictions for regression
from sklearn.datasets import make_regression
from sklearn.ensemble import ExtraTreesRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=3)
# define the model
model = ExtraTreesRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [-0.56996683, 0.80144889, 2.77523539, 1.32554027, -1.44494378, -0.80834175,
    -0.84142896, 0.57710245, 0.96235932, -0.66303907, -1.13994112, 0.49887995, 1.40752035,
    -0.2995842, -0.05708706, -2.08701456, 1.17768469, 0.13474234, 0.09518152, -0.07603207]
yhat = model.predict([row])
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Listing 18.11: Example of making a prediction with the extra trees ensemble on the synthetic regression dataset.

Running the example fits the Extra Trees ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: 53
```

Listing 18.12: Example output from making a prediction with the extra trees ensemble on the synthetic regression dataset.

Now that we are familiar with using the scikit-learn API to evaluate and use Extra Trees ensembles, let's look at configuring the model.

18.4 Extra Trees Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the Extra Trees ensemble and their effect on model performance.

18.4.1 Explore Number of Trees

An important hyperparameter for the Extra Trees algorithm is the number of decision trees used in the ensemble. Typically, the number of trees is increased until the model performance stabilizes. Intuition might suggest that more trees will lead to overfitting, although this is not the case. Bagging, Random Forest, and Extra Trees algorithms appear to be somewhat immune to overfitting the training dataset given the stochastic nature of the learning algorithm. The number of trees can be set via the `n_estimators` argument and defaults to 100. The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore extra trees number of trees effect on performance
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.ensemble import ExtraTreesClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=4)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define number of trees to consider
  n_trees = [10, 50, 100, 500, 1000, 5000]
  for n in n_trees:
    models[str(n)] = ExtraTreesClassifier(n_estimators=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
```

```
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 18.13: Example of exploring the effect of the number of trees on the extra trees ensemble.

Running the example first reports the mean accuracy for each configured number of decision trees.

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

In this case, we can see that performance rises and stays flat after about 100 trees. Mean accuracy scores fluctuate across 100, 500, and 1,000 trees and beyond and this may be statistical noise.

```
>10 0.860 (0.029)
>50 0.904 (0.027)
>100 0.908 (0.026)
>500 0.910 (0.027)
>1000 0.910 (0.026)
>5000 0.912 (0.026)
```

Listing 18.14: Example output from exploring the effect of the number of trees on the extra trees ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees. We can see the general trend of increasing performance with the number of trees, perhaps leveling out after 100 trees.

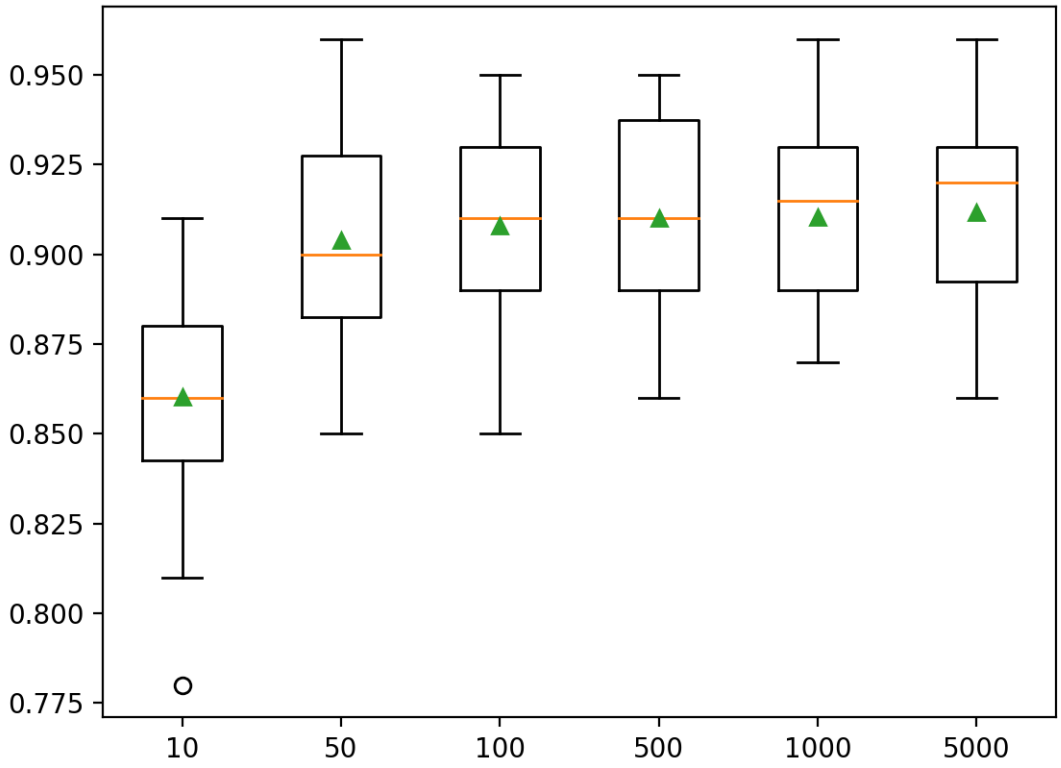

Figure 18.1: Box Plot of Extra Trees Ensemble Size vs. Classification Accuracy.

18.4.2 Explore Number of Features

The number of features that is randomly sampled for each split point is perhaps the most important feature to configure for Extra Trees, as it is for Random Forest. Like Random Forest, the Extra Trees algorithm is not sensitive to the specific value used, although it is an important hyperparameter to tune. It is set via the `max_features` argument and defaults to the square root of the number of input features. In this case for our synthetic dataset, this would be $\sqrt{20}$ or about four features. The example below explores the effect of the number of features randomly selected at each split point on model accuracy. We will try values from 1 to 20 and would expect a small value around four to perform well based on the heuristic.

```
# explore extra trees number of features effect on performance
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.ensemble import ExtraTreesClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
```

```
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=4)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore number of features from 1 to 20
  for i in range(1, 21):
    models[str(i)] = ExtraTreesClassifier(max_features=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 18.15: Example of exploring the effect of the number of features on the extra trees ensemble.

Running the example first reports the mean accuracy for each feature set size.

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

In this case, the results suggest that a value between four and nine would be appropriate, confirming the sensible default of four on this dataset. A value of nine might even be better given the larger mean and smaller standard deviation in classification accuracy, although the differences in scores may or may not be statistically significant.

```
>1 0.901 (0.028)
>2 0.909 (0.028)
>3 0.901 (0.026)
```

```
>4 0.909 (0.030)
>5 0.909 (0.028)
>6 0.910 (0.025)
>7 0.908 (0.030)
>8 0.907 (0.025)
>9 0.912 (0.024)
>10 0.904 (0.029)
>11 0.904 (0.025)
>12 0.908 (0.026)
>13 0.908 (0.026)
>14 0.906 (0.030)
>15 0.909 (0.024)
>16 0.908 (0.023)
>17 0.910 (0.021)
>18 0.909 (0.023)
>19 0.907 (0.025)
>20 0.903 (0.025)
```

Listing 18.16: Example output from exploring the effect of the number of features on the extra trees ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each feature set size. We see a trend in performance rising and peaking with values between four and nine and falling or staying flat as larger feature set sizes are considered.

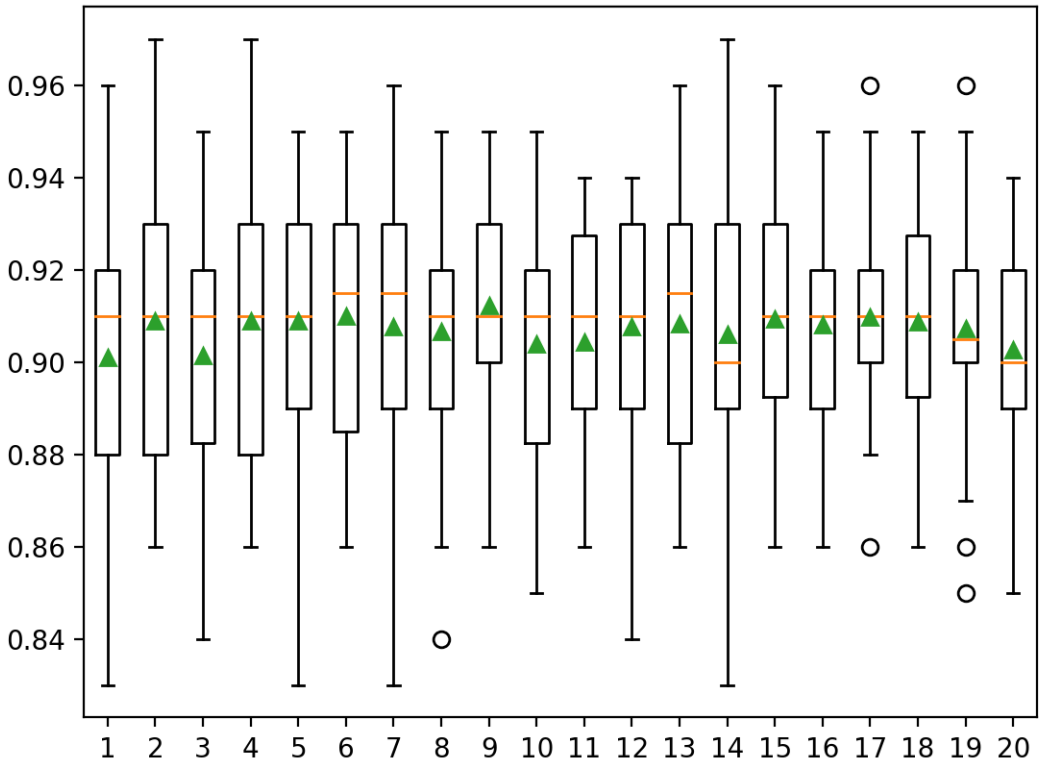

Figure 18.2: Box Plot of Extra Trees Feature Set Size vs. Classification Accuracy.

18.4.3 Explore Minimum Samples per Split

A final interesting hyperparameter is the number of samples in a node of the decision tree before adding a split. New splits are only added to a decision tree if the number of samples is equal to or exceeds this value. It is set via the `min_samples_split` argument and defaults to two samples (the lowest value). Smaller numbers of samples result in more splits and a deeper, more specialized tree. In turn, this can mean lower correlation between the predictions made by trees in the ensemble and potentially lift performance. The example below explores the effect of Extra Trees minimum samples before splitting on model performance, test values between two and 14.

```
# explore extra trees minimum number of samples for a split effect on performance
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.ensemble import ExtraTreesClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
```

```
      n_redundant=5, random_state=4)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore the number of samples per split from 2 to 14
  for i in range(2, 15):
    models[str(i)] = ExtraTreesClassifier(min_samples_split=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 18.17: Example of exploring the effect of the minimum number of samples per split on the extra trees ensemble.

Running the example first reports the mean accuracy for each configured maximum tree depth.

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

In this case, we can see that small values result in better performance, confirming the sensible default of two.

```
>2 0.909 (0.025)
>3 0.907 (0.026)
>4 0.907 (0.026)
>5 0.902 (0.028)
>6 0.902 (0.027)
```

```
>7 0.904 (0.024)
>8 0.899 (0.026)
>9 0.896 (0.029)
>10 0.896 (0.027)
>11 0.897 (0.028)
>12 0.894 (0.026)
>13 0.890 (0.026)
>14 0.892 (0.027)
```

Listing 18.18: Example output from exploring the effect of the minimum number of samples per split on the extra trees ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured minimum samples before splitting. In this case, we can see a trend of improved performance with fewer minimum samples for a split, as we might expect.

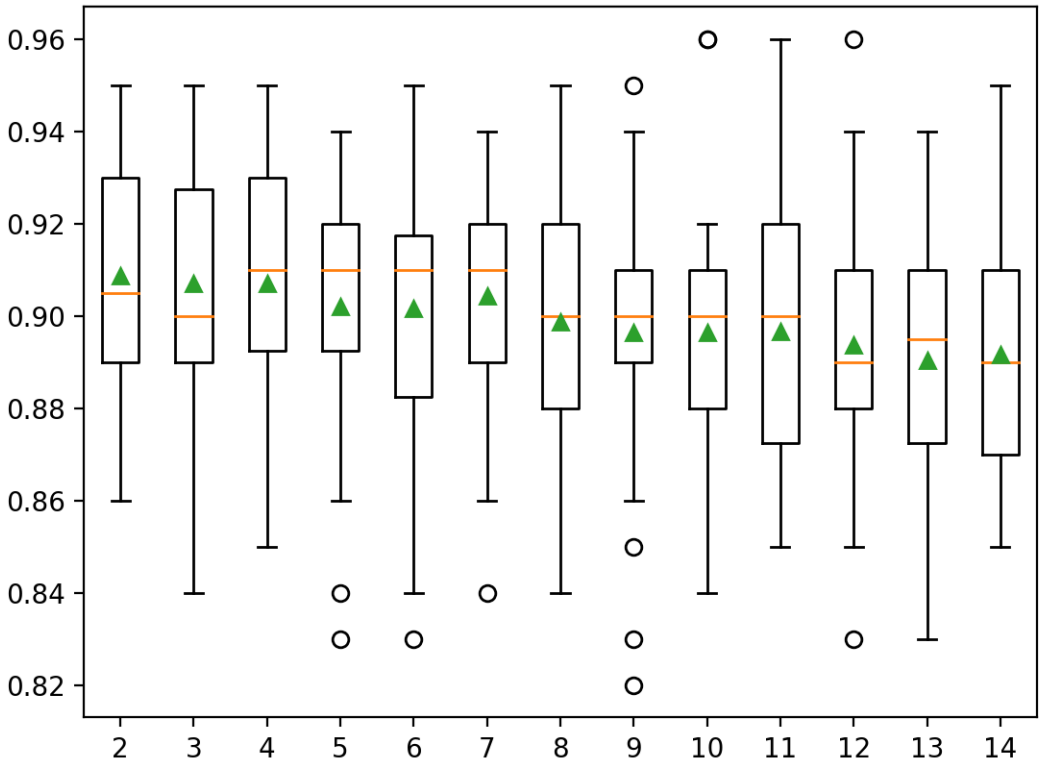

Figure 18.3: Box Plot of Extra Trees Minimum Samples per Split vs. Classification Accuracy.

18.5 Further Reading

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

Papers

- *Extremely Randomized Trees*, 2006.
  https://link.springer.com/article/10.1007/s10994-006-6226-1

APIs

- `sklearn.ensemble.ExtraTreesClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html
- `sklearn.ensemble.ExtraTreesRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesRegressor.html

18.6 Summary

In this tutorial, you discovered how to develop Extra Trees ensembles for classification and regression. Specifically, you learned:

- Extra Trees ensemble is an ensemble of decision trees and is related to bagging and random forest.
- How to use the Extra Trees ensemble for classification and regression with scikit-learn.
- How to explore the effect of Extra Trees model hyperparameters on model performance.

Next

In the next section, we will take a look at developing a custom bagging ensemble comprised of models fit with different data preparation techniques.

Chapter 19

Data Transform Bagging Ensemble

Bootstrap aggregation, or bagging, is an ensemble where each model is trained on a different sample of the training dataset. The idea of bagging can be generalized to other techniques for changing the training dataset and fitting the same model on each changed version of the data. One approach is to use data transforms that change the scale and probability distribution of input variables as the basis for the training of contributing members to a bagging-like ensemble. We can refer to this as data transform bagging or a data transform ensemble. In this tutorial, you will discover how to develop a data transform ensemble. After completing this tutorial, you will know:

- Data transforms can be used as the basis for a bagging-type ensemble where the same model is trained on different views of a training dataset.
- How to develop a data transform ensemble for classification and confirm the ensemble performs better than any contributing member.
- How to develop and evaluate a data transform ensemble for regression predictive modeling.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Data Transform Bagging
2. Data Transform Ensemble for Classification
3. Data Transform Ensemble for Regression

19.2 Data Transform Bagging

Bootstrap aggregation, or bagging for short, is an ensemble learning technique based on the idea of fitting the same model type on multiple different samples of the same training dataset. The hope is that small differences in the training dataset used to fit each model will result in small

differences in the capabilities of models. For ensemble learning, this is referred to as diversity of ensemble members and is intended to de-correlate the predictions (or prediction errors) made by each contributing member. Although it was designed to be used with decision trees and each data sample is made using the bootstrap method (selection with replacement), the approach has spawned a whole subfield of study with hundreds of variations on the approach.

We can construct our own bagging ensembles by changing the dataset used to train each contributing member in new and unique ways. One approach would be to apply a different data preparation transform to the dataset for each contributing ensemble member. This is based on the premise that we cannot know the representational form for a training dataset that exposes the unknown underlying structure of the dataset to the learning algorithms. This motivates the need to evaluate models with a suite of different data transforms, such as changing the scale and probability distribution, in order to discover what works.

This approach can be used where a suite of different transforms of the same training dataset is created, a model trained on each, and the predictions combined using simple statistics such as averaging. For lack of a better name, we will refer to this as *Data Transform Bagging* or a *Data Transform Ensemble*. There are many transforms that we can use, but perhaps a good starting point would be a selection that changes the scale and probability distribution, such as:

- Normalization (fixed range).
- Standardization (zero mean).
- Robust Standardization (robust to outliers).
- Power Transform (remove skew).
- Quantile Transform (change distribution).
- Discretization (k-bins).

The approach is likely to be more effective when used with a base-model that trains different or very different models based on the effects of the data transform. Changing the scale of the distribution may only be appropriate with models that are sensitive to changes in the scale of input variables, such as those that calculate a weighted sum, such as logistic regression and neural networks, and those that use distance measures, such as k-nearest neighbors and support vector machines. However, changes to the probability distribution for input variables would likely impact most machine learning models. Now that we are familiar with the approach, let's explore how we can develop a data transform ensemble for classification problems.

19.3 Data Transform Ensemble for Classification

We can develop a data transform approach to bagging for classification using the scikit-learn library. The library provides a suite of standard transforms that we can use directly. Each ensemble member can be defined as a `Pipeline`, with the transform followed by the predictive model, in order to avoid any data leakage and, in turn, produce optimistic results. Finally, a voting ensemble can be used to combine the predictions from each pipeline. First, we can define a synthetic binary classification dataset as the basis for exploring this type of ensemble. The example below creates a dataset with 1,000 examples each comprising 20 input features, where 15 of them contain information for predicting the target.

```
# synthetic 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=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 19.1: Example of creating the synthetic classification dataset.

Running the example will create the dataset and summarizes the shape of the data arrays, confirming our expectations.

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

Listing 19.2: Example output from creating the synthetic classification dataset.

Next, we establish a baseline on the problem using the predictive model we intend to use in our ensemble. It is standard practice to use a decision tree in bagging ensembles, so in this case, we will use the `DecisionTreeClassifier` with default hyperparameters. We will evaluate the model using standard practices, in this case, repeated stratified k-fold cross-validation with three repeats and 10 folds. The performance will be reported using the mean of the classification accuracy across all folds and repeats. The complete example of evaluating a decision tree on the synthetic classification dataset is listed below.

```
# evaluate decision tree on synthetic classification dataset
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.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1)
# define the model
model = DecisionTreeClassifier()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 19.3: Example of evaluating a baseline model on the classification dataset.

Running the example reports the mean classification accuracy of the decision tree on the synthetic classification dataset.

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

In this case, we can see that the model achieved a classification accuracy of about 82.3 percent. This score provides a baseline in performance from which we expect a data transform ensemble to improve upon.

```
Mean Accuracy: 0.823 (0.039)
```

Listing 19.4: Example output from evaluating a baseline model on the classification dataset.

Next, we can develop an ensemble of decision trees, each fit on a different transform of the input data. First, we can define each ensemble member as a modeling pipeline. The first step will be the data transform and the second will be a decision tree classifier.

For example, the pipeline for a normalization transform with the `MinMaxScaler` class would look as follows:

```
...
# normalization
norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeClassifier())])
```

Listing 19.5: Example of defining a modeling pipeline.

We can repeat this for each transform or transform configuration that we want to use and add all of the model pipelines to a list. The `VotingClassifier` class can be used to combine the predictions from all of the models. The voting ensemble simply averages the predictions made by contributing members. We will go into a lot more detail into how voting ensembles work in Chapter 25. This class takes an `estimators` argument that is a list of tuples where each tuple has a name and the model or modeling pipeline. For example:

```
...
# normalization
norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeClassifier())])
models.append(('norm', norm))
...
# define the voting ensemble
ensemble = VotingClassifier(estimators=models, voting='hard')
```

Listing 19.6: Example of defining a voting ensemble of modeling pipelines.

To make the code easier to read, we can define a function `get_ensemble()` to create the members and data transform ensemble itself.

```
# get a voting ensemble of models
def get_ensemble():
  # define the base models
  models = list()
  # normalization
  norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeClassifier())])
  models.append(('norm', norm))
  # standardization
  st = Pipeline([('s', StandardScaler()), ('m', DecisionTreeClassifier())])
  models.append(('std', st))
  # robust
  robust = Pipeline([('s', RobustScaler()), ('m', DecisionTreeClassifier())])
  models.append(('robust', robust))
  # power
  power = Pipeline([('s', PowerTransformer()), ('m', DecisionTreeClassifier())])
  models.append(('power', power))
  # quantile
  quant = Pipeline([('s', QuantileTransformer(n_quantiles=100,
      output_distribution='normal')), ('m', DecisionTreeClassifier())])
  models.append(('quant', quant))
```

```
  # kbins
  kbins = Pipeline([('s', KBinsDiscretizer(n_bins=20, encode='ordinal')), ('m',
      DecisionTreeClassifier())])
  models.append(('kbins', kbins))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  return ensemble
```

Listing 19.7: Example of function for defining a data preparation pipeline ensemble for classification.

We can then call this function and evaluate the voting ensemble as per normal, just like we did for the decision tree above. Tying this together, the complete example is listed below.

```
# evaluate data transform bagging ensemble on a classification dataset
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.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import QuantileTransformer
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline

# get a voting ensemble of models
def get_ensemble():
  # define the base models
  models = list()
  # normalization
  norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeClassifier())])
  models.append(('norm', norm))
  # standardization
  st = Pipeline([('s', StandardScaler()), ('m', DecisionTreeClassifier())])
  models.append(('std', st))
  # robust
  robust = Pipeline([('s', RobustScaler()), ('m', DecisionTreeClassifier())])
  models.append(('robust', robust))
  # power
  power = Pipeline([('s', PowerTransformer()), ('m', DecisionTreeClassifier())])
  models.append(('power', power))
  # quantile
  quant = Pipeline([('s', QuantileTransformer(n_quantiles=100,
      output_distribution='normal')), ('m', DecisionTreeClassifier())])
  models.append(('quant', quant))
  # kbins
  kbins = Pipeline([('s', KBinsDiscretizer(n_bins=20, encode='ordinal')), ('m',
      DecisionTreeClassifier())])
  models.append(('kbins', kbins))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
```

```
  return ensemble

# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1)
# get models
ensemble = get_ensemble()
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 19.8: Example of evaluating a data preparation ensemble on the classification dataset.

Running the example reports the mean classification accuracy of the data transform ensemble on the synthetic classification dataset.

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

In this case, we can see that the data transform ensemble achieved a classification accuracy of about 83.8 percent, which is a lift over using a decision tree alone that achieved an accuracy of about 82.3 percent.

```
Mean Accuracy: 0.838 (0.042)
```

Listing 19.9: Example output from evaluating a data preparation ensemble on the classification dataset.

Although the ensemble performed well compared to a single decision tree, a limitation of this test is that we do not know if the ensemble performed better than any contributing member. This is important, as if a contributing member to the ensemble performs better, then it would be simpler and easier to use the member itself as the model instead of the ensemble. We can check this by evaluating the performance of each individual model and comparing the results to the ensemble. First, we can update the `get_ensemble()` function to return a list of models to evaluate composed of the individual ensemble members as well as the ensemble itself.

```
# get a voting ensemble of models
def get_ensemble():
  # define the base models
  models = list()
  # normalization
  norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeClassifier())])
  models.append(('norm', norm))
  # standardization
  st = Pipeline([('s', StandardScaler()), ('m', DecisionTreeClassifier())])
  models.append(('std', st))
  # robust
  robust = Pipeline([('s', RobustScaler()), ('m', DecisionTreeClassifier())])
  models.append(('robust', robust))
  # power
  power = Pipeline([('s', PowerTransformer()), ('m', DecisionTreeClassifier())])
```

```
  models.append(('power', power))
  # quantile
  quant = Pipeline([('s', QuantileTransformer(n_quantiles=100,
      output_distribution='normal')), ('m', DecisionTreeClassifier())])
  models.append(('quant', quant))
  # kbins
  kbins = Pipeline([('s', KBinsDiscretizer(n_bins=20, encode='ordinal')), ('m',
      DecisionTreeClassifier())])
  models.append(('kbins', kbins))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  # return a list of tuples each with a name and model
  return models + [('ensemble', ensemble)]
```

Listing 19.10: Example of a function for defining a data preparation pipeline ensemble and standalone models.

We can call this function and enumerate each model, evaluating it, reporting the performance, and storing the results.

```
...
# get models
models = get_ensemble()
# evaluate each model
results = list()
for name,model in models:
  # define the evaluation method
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model on the dataset
  n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # report performance
  print('>%s: %.3f (%.3f)' % (name, mean(n_scores), std(n_scores)))
  results.append(n_scores)
```

Listing 19.11: Example of evaluating ensemble members as standalone models.

Finally, we can plot the distribution of accuracy scores as box and whisker plots side by side and compare the distribution of scores directly. Visually, we would hope that the spread of scores for the ensemble skews higher than any individual member and that the central tendency of the distribution (mean and median) are also higher than any member.

```
...
# plot the results for comparison
pyplot.boxplot(results, labels=[n for n,_ in models], showmeans=True)
pyplot.show()
```

Listing 19.12: Example of plotting the distribution of results for each evaluated model.

Tying this together, the complete example of comparing the performance of contributing members to the performance of the data transform ensemble is listed below.

```
# comparison of data transform ensemble to each contributing member for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
```

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import QuantileTransformer
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline
from matplotlib import pyplot

# get a voting ensemble of models
def get_ensemble():
  # define the base models
  models = list()
  # normalization
  norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeClassifier())])
  models.append(('norm', norm))
  # standardization
  st = Pipeline([('s', StandardScaler()), ('m', DecisionTreeClassifier())])
  models.append(('std', st))
  # robust
  robust = Pipeline([('s', RobustScaler()), ('m', DecisionTreeClassifier())])
  models.append(('robust', robust))
  # power
  power = Pipeline([('s', PowerTransformer()), ('m', DecisionTreeClassifier())])
  models.append(('power', power))
  # quantile
  quant = Pipeline([('s', QuantileTransformer(n_quantiles=100,
      output_distribution='normal')), ('m', DecisionTreeClassifier())])
  models.append(('quant', quant))
  # kbins
  kbins = Pipeline([('s', KBinsDiscretizer(n_bins=20, encode='ordinal')), ('m',
      DecisionTreeClassifier())])
  models.append(('kbins', kbins))
  # define the voting ensemble
  ensemble = VotingClassifier(estimators=models, voting='hard')
  # return a list of tuples each with a name and model
  return models + [('ensemble', ensemble)]

# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1)
# get models
models = get_ensemble()
# evaluate each model
results = list()
for name, model in models:
  # define the evaluation method
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model on the dataset
  n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # store the results
  results.append(n_scores)
  # report performance
  print('>%s: %.3f (%.3f)' % (name, mean(n_scores), std(n_scores)))
```

```
# plot the results for comparison
pyplot.boxplot(results, labels=[n for n,_ in models], showmeans=True)
pyplot.show()
```

Listing 19.13: Example of comparing a data preparation ensemble to standalone models on the classification dataset.

Running the example first reports the mean and standard classification accuracy of each individual model, ending with the performance of the ensemble that combines the models.

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

In this case, we can see that a number of the individual members perform well, such as *kbins* that achieves an accuracy of about 83.3 percent, and *std* that achieves an accuracy of about 83.1 percent. We can also see that the ensemble achieves better overall performance compared to any contributing member, with an accuracy of about 83.4 percent.

```
>norm: 0.821 (0.041)
>std: 0.831 (0.045)
>robust: 0.826 (0.044)
>power: 0.825 (0.045)
>quant: 0.817 (0.042)
>kbins: 0.833 (0.035)
>ensemble: 0.834 (0.040)
```

Listing 19.14: Example output from comparing a data preparation ensemble to standalone models on the classification dataset.

A figure is also created showing box and whisker plots of classification accuracy for each individual model as well as the data transform ensemble. We can see that the distribution for the ensemble is skewed up, which is what we might hope, and that the mean (green triangle) is slightly higher than those of the individual ensemble members.

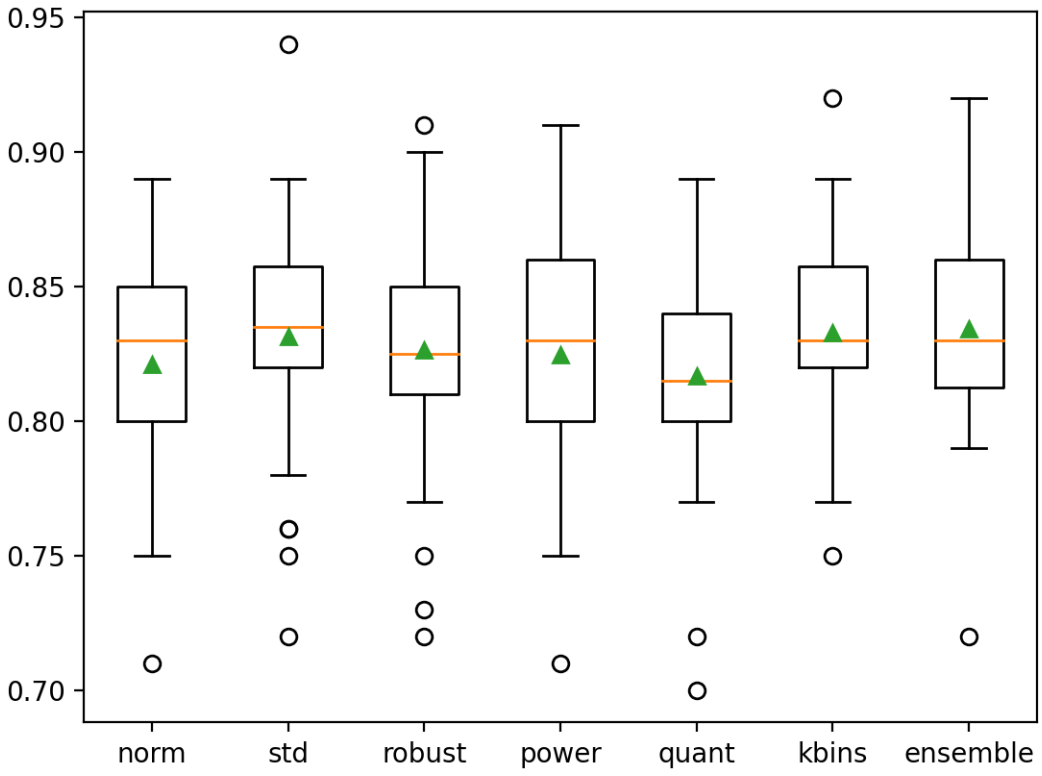

Figure 19.1: Box and Whisker Plot of Accuracy Distribution for Individual Models and Data Transform Ensemble.

Now that we are familiar with how to develop a data transform ensemble for classification, let's look at doing the same for regression.

19.4 Data Transform Ensemble for Regression

In this section, we will explore developing a data transform ensemble for a regression predictive modeling problem. First, we can define a synthetic binary regression dataset as the basis for exploring this type of ensemble. The example below creates a dataset with 1,000 examples each of 100 input features where 10 of them contain information for predicting the target.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 19.15: Example of creating the synthetic regression dataset.

Running the example creates the dataset and confirms the data has the expected shape.

```
(1000, 100) (1000,)
```

Listing 19.16: Example output from creating the synthetic regression dataset.

Next, we can establish a baseline in performance on the synthetic dataset by fitting and evaluating the base-model that we intend to use in the ensemble, in this case, a `DecisionTreeRegressor`. The model will be evaluated using repeated k-fold cross-validation with three repeats and 10 folds. Model performance on the dataset will be reported using the mean absolute error, or MAE. The example below evaluates the decision tree on the synthetic regression dataset.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate decision tree on synthetic regression dataset
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.tree import DecisionTreeRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# define the model
model = DecisionTreeRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 19.17: Example of evaluating a baseline model on the regression dataset.

Running the example reports the MAE of the decision tree on the synthetic regression dataset.

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

In this case, we can see that the model achieved a MAE of about 139.817. This provides a floor in performance that we expect the ensemble model to improve upon.

```
MAE: -139.817 (12.449)
```

Listing 19.18: Example output from evaluating a baseline model on the regression dataset.

Next, we can develop and evaluate the ensemble. We will use the same data transforms from the previous section. The `VotingRegressor` will be used to combine the predictions, which is appropriate for regression problems. The `get_ensemble()` function defined below creates the individual models and the ensemble model and combines all of the models as a list of tuples for evaluation.

```
# get a voting ensemble of models
def get_ensemble():
  # define the base models
  models = list()
  # normalization
  norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeRegressor())])
  models.append(('norm', norm))
  # standardization
  st = Pipeline([('s', StandardScaler()), ('m', DecisionTreeRegressor())])
  models.append(('std', st))
  # robust
  robust = Pipeline([('s', RobustScaler()), ('m', DecisionTreeRegressor())])
  models.append(('robust', robust))
  # power
  power = Pipeline([('s', PowerTransformer()), ('m', DecisionTreeRegressor())])
  models.append(('power', power))
  # quantile
  quant = Pipeline([('s', QuantileTransformer(n_quantiles=100,
      output_distribution='normal')), ('m', DecisionTreeRegressor())])
  models.append(('quant', quant))
  # kbins
  kbins = Pipeline([('s', KBinsDiscretizer(n_bins=20, encode='ordinal')), ('m',
      DecisionTreeRegressor())])
  models.append(('kbins', kbins))
  # define the voting ensemble
  ensemble = VotingRegressor(estimators=models)
  # return a list of tuples each with a name and model
  return models + [('ensemble', ensemble)]
```

Listing 19.19: Example of a function for defining a data preparation pipeline ensemble and standalone models for regression.

We can then call this function and evaluate each contributing modeling pipeline independently and compare the results to the ensemble of the pipelines. Our expectation, as before, is that the ensemble results in a lift in performance over any individual model. If it does not, then the top-performing individual model should be chosen instead. Tying this together, the complete example for evaluating a data transform ensemble for a regression dataset is listed below.

```
# comparison of data transform ensemble to each contributing member for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import QuantileTransformer
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor
from sklearn.pipeline import Pipeline
from matplotlib import pyplot
```

```
# get a voting ensemble of models
def get_ensemble():
  # define the base models
  models = list()
  # normalization
  norm = Pipeline([('s', MinMaxScaler()), ('m', DecisionTreeRegressor())])
  models.append(('norm', norm))
  # standardization
  st = Pipeline([('s', StandardScaler()), ('m', DecisionTreeRegressor())])
  models.append(('std', st))
  # robust
  robust = Pipeline([('s', RobustScaler()), ('m', DecisionTreeRegressor())])
  models.append(('robust', robust))
  # power
  power = Pipeline([('s', PowerTransformer()), ('m', DecisionTreeRegressor())])
  models.append(('power', power))
  # quantile
  quant = Pipeline([('s', QuantileTransformer(n_quantiles=100,
      output_distribution='normal')), ('m', DecisionTreeRegressor())])
  models.append(('quant', quant))
  # kbins
  kbins = Pipeline([('s', KBinsDiscretizer(n_bins=20, encode='ordinal')), ('m',
      DecisionTreeRegressor())])
  models.append(('kbins', kbins))
  # define the voting ensemble
  ensemble = VotingRegressor(estimators=models)
  # return a list of tuples each with a name and model
  return models + [('ensemble', ensemble)]

# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=100, n_informative=10, noise=0.1,
    random_state=1)
# get models
models = get_ensemble()
# evaluate each model
results = list()
for name, model in models:
  # define the evaluation method
  cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model on the dataset
  n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv,
      n_jobs=-1)
  # store results
  results.append(n_scores)
  # report performance
  print('>%s: %.3f (%.3f)' % (name, mean(n_scores), std(n_scores)))
# plot the results for comparison
pyplot.boxplot(results, labels=[n for n,_ in models], showmeans=True)
pyplot.show()
```

Listing 19.20: Example of comparing a data preparation ensemble to standalone models on the regression dataset.

Running the example first reports the MAE of each individual model, ending with the performance of the ensemble that combines the models.

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

We can see that each model performs about the same, with MAE error scores around 140, all better than the decision tree used in isolation. Interestingly, the ensemble performs the best, out-performing all of the individual members and the tree with no transforms, achieving a MAE of about 126.487. This result suggests that although each pipeline performs worse than a single tree without transforms, each pipeline is making different errors and that the average of the models is able to leverage and harness these differences toward lower error.

```
>norm: -140.559 (11.783)
>std: -140.582 (11.996)
>robust: -140.813 (11.827)
>power: -141.089 (12.668)
>quant: -141.109 (11.097)
>kbins: -145.134 (11.638)
>ensemble: -126.487 (9.999)
```

Listing 19.21: Example output from comparing a data preparation ensemble to standalone models on the regression dataset.

A figure is created comparing the distribution of MAE scores for each pipeline and the ensemble. As we hoped, the distribution for the ensemble skews higher compared to all of the other models and has a higher (smaller) central tendency (mean and median indicated by the green triangle and orange line respectively).

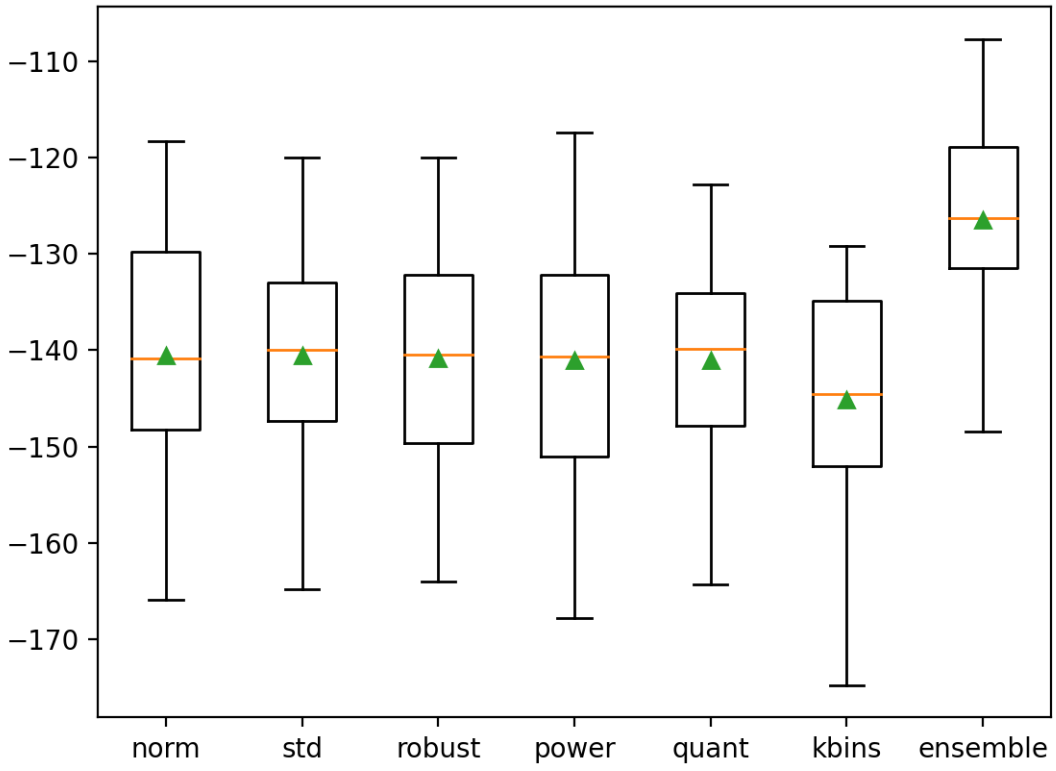

Figure 19.2: Box and Whisker Plot of MAE Distributions for Individual Models and Data Transform Ensemble.

19.5 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

APIs

- `sklearn.ensemble.VotingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

- `sklearn.ensemble.VotingRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingRegressor.html

19.6 Summary

In this tutorial, you discovered how to develop a data transform ensemble. Specifically, you learned:

- Data transforms can be used as the basis for a bagging-type ensemble where the same model is trained on different views of a training dataset.
- How to develop a data transform ensemble for classification and confirm the ensemble performs better than any contributing member.
- How to develop and evaluate a data transform ensemble for regression predictive modeling.

Next

This was the final tutorial in this part, in the next part we will take a closer look at boosting ensemble algorithms.

Part VI

Boosting

Chapter 20

Strong vs Weak Learners

It is common to describe ensemble learning techniques in terms of weak and strong learners. For example, we may desire to construct a strong learner from the predictions of many weak learners. In fact, this is the explicit goal of the boosting class of ensemble learning algorithms. Although we may describe models as weak or strong generally, the terms have a specific formal definition and are used as the basis for an important finding from the field of computational learning theory. In this tutorial, you will discover weak and strong learners and their relationship with ensemble learning. After completing this tutorial, you will know:

- Weak learners are models that perform slightly better than random guessing.
- Strong learners are models that have arbitrarily good accuracy.
- Weak and strong learners are tools from computational learning theory and provide the basis for the development of the boosting class of ensemble methods.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Weak Learners
2. Strong Learners
3. Weak vs. Strong Learners and Boosting

20.2 Weak Learners

A weak classifier is a model for binary classification that performs slightly better than random guessing.

> A weak learner produces a classifier which is only slightly more accurate than random classification.

— Page 21, *Pattern Classification Using Ensemble Methods*, 2010.

This means that the model will make predictions that are known to have some skill, e.g. making the capabilities of the model weak, although not so weak that the model has no skill, e.g. performs worse than random.

- **Weak Classifier**: Formally, a classifier that achieves slightly better than 50 percent accuracy.

A weak classifier is sometimes called a *weak learner* or *base learner* and the concept can be generalized beyond binary classification. Although the concept of a weak learner is well understood in the context of binary classification, it can be taken colloquially to mean any model that performs slightly better than a naive prediction method. In this sense, it is a useful tool for thinking about the capability of classifiers and the composition of ensembles.

- **Weak Learner**: Colloquially, a model that performs slightly better than a naive model.

More formally, the notion has been generalized to multiclass classification and has a different meaning beyond better than 50 percent accuracy.

> For binary classification, it is well known that the exact requirement for weak learners is to be better than random guess. [...] Notice that requiring base learners to be better than random guess is too weak for multi-class problems, yet requiring better than 50% accuracy is too stringent.

— Page 46, *Ensemble Methods*, 2012.

It is based on formal computational learning theory that proposes a class of learning methods that possess weakly learnability, meaning that they perform better than random guessing. Weak learnability is proposed as a simplification of the more desirable strong learnability, where a learner achieves arbitrary good classification accuracy.

> A weaker model of learnability, called weak learnability, drops the requirement that the learner be able to achieve arbitrarily high accuracy; a weak learning algorithm needs only output an hypothesis that performs slightly better (by an inverse polynomial) than random guessing.

— *The Strength of Weak Learnability*, 1990.

It is a useful concept as it is often used to describe the capabilities of contributing members of ensemble learning algorithms. For example, sometimes members of a bootstrap aggregation are referred to as weak learners as opposed to strong, at least in the colloquial meaning of the term. More specifically, weak learners are the basis for the boosting class of ensemble learning algorithms.

> The term boosting refers to a family of algorithms that are able to convert weak learners to strong learners.

— Page 23, *Ensemble Methods*, 2012.

The most commonly used type of weak learning model is the decision tree. This is because the weakness of the tree can be controlled by the depth of the tree during construction. The weakest decision tree consists of a single node that makes a decision on one input variable and outputs a binary prediction, for a binary classification task. This is generally referred to as a *decision stump*.

> Here the weak classifier is just a "stump": a two terminal-node classification tree.
>
> — Page 339, *The Elements of Statistical Learning*, 2016.

It is used as a weak learner so often that decision stump and weak learner are practically synonyms.

- **Decision Stump**: A decision tree with a single node operating on one input variable, the output of which makes a prediction directly.

Nevertheless, other models can also be configured to be weak learners.

> Because boosting requires a weak learner, almost any technique with tuning parameters can be made into a weak learner. Trees, as it turns out, make an excellent base learner for boosting ...
>
> — Page 205, *Applied Predictive Modeling*, 2013.

Although not formally known as weak learners, we can consider the following as candidate weak learning models:

- ***k*-Nearest Neighbors**, with k=1 operating on one or a subset of input variables.
- **Multilayer Perceptron**, with a single node operating on one or a subset of input variables.
- **Naive Bayes**, operating on a single input variable.

Now that we are familiar with a weak learner, let's take a closer look at strong learners.

20.3 Strong Learners

A strong classifier is a model for binary classification that performs with arbitrary performance, much better than random guessing.

> A class of concepts is learnable (or strongly learnable) if there exists a polynomial-time algorithm that achieves low error with high confidence for all concepts in the class.
>
> — *The Strength of Weak Learnability*, 1990.

This is sometimes interpreted to mean perfect skill on a training or holdout dataset, although more likely refers to a *good* or *usefully skillful* model.

- **Strong Classifier**: Formally, a classifier that achieves arbitrarily good accuracy.

We seek strong classifiers for predictive modeling problems. It is the goal of the modeling project to develop a strong classifier that makes mostly correct predictions with high confidence. Again, although the concept of a strong classifier is well understood for binary classification, it can be generalized to other problem types and we can interpret the concept less formally as a well-performing model, perhaps near-optimal.

- **Strong Learner**: Colloquially, a model that performs very well compared to a naive model.

We are attempting to develop a strong model when we fit a machine learning model directly on a dataset. For example, we might consider the following algorithms as techniques for fitting a strong model in the colloquial sense, where the hyperparameters of each method are tuned for the target problem:

- Logistic Regression.
- Support Vector Machine.
- k-Nearest Neighbors.

Strong learning is what we seek, and we can contrast their capability with weak learners, although we can also construct strong learners from weak learners.

20.4 Weak vs. Strong Learners and Boosting

We have established that weak learners perform slightly better than random, and that strong learners are good or even near-optimal and it is the latter that we seek for a predictive modeling project. In computational learning theory, specifically Probably Approximately Correct (PAC) learning, the formal classes of weak and strong learnability were defined with the open question as to whether the two were equivalent or not.

> The proof presented here is constructive; an explicit method is described for directly converting a weak learning algorithm into one that achieves arbitrary accuracy. The construction uses filtering to modify the distribution of examples in such a way as to force the weak learning algorithm to focus on the harder-to-learn parts of the distribution.
>
> — *The Strength of Weak Learnability*, 1990.

Later, it was discovered that they are indeed equivalent. More so that a strong learner can be constructed from many weak learners, formally defined. This provided the basis for the boosting class of ensemble learning methods.

> The main result is a proof of the perhaps surprising equivalence of strong and weak learnability.

— *The Strength of Weak Learnability*, 1990.

Although this theoretical finding was made, it still took years before the first viable boosting methods were developed, implementing the procedure. Most notably Adaptive Boosting, referred to as AdaBoost, was the first successful boosting method, later leading to a large number of methods, culminating today in highly successful techniques such as gradient boosting and implementations such as Extreme Gradient Boosting (XGBoost).

> Ensembles of weak learners was mostly studied in the machine learning community. In this thread, researchers often work on weak learners and try to design powerful algorithms to boost the performance from weak to strong. This thread of work has led to the birth of famous ensemble methods such as AdaBoost, Bagging, etc., and theoretical understanding on why and how weak learners can be boosted to strong ones.

— Page 16, *Ensemble Methods*, 2012.

Generally, the goal of boosting ensembles is to develop a large number of weak learners for a predictive learning problem, then best combine them in order to achieve a strong learner. This is a good goal as weak learners are easy to prepare but not desirable, and strong learners are hard to prepare and highly desirable.

> Since strong learners are desirable yet difficult to get, while weak learners are easy to obtain in real practice, this result opens a promising direction of generating strong learners by ensemble methods.

— Pages 16–17, *Ensemble Methods*, 2012.

- **Weak Learner**: Easy to prepare, but not desirable due to their low skill.
- **Strong Learner**: Hard to prepare, but desirable because of their high skill.

The procedure that was found to achieve this is to sequentially develop weak learners and add them to the ensemble, where each weak learner is trained in a way to pay more attention to parts of the problem domain that prior models got wrong. Although all boosting techniques follow this general procedure with specific differences and optimizations, the notion of weak and strong learners is a useful concept more generally for machine learning and ensemble learning. For example, we have already seen that we can describe the goal of a predictive model as: *develop a strong model*. It is common practice to evaluate the performance of a model against a baseline or naive model, such as random predictions for binary classification. A weak learner is very much like the naive model, although slightly skillful and using a minimum of information from the problem domain, as opposed to completely naive.

Consider that although we do not technically construct weak learners in bootstrap aggregation (bagging, introduced in Chapter 14), meaning the members are not decision stumps, we do aim to create weaker decision trees to comprise the ensemble. This is often achieved by fitting the trees on sampled subsets of the data and not pruning the trees, allowing them to overfit the training data slightly.

> For classification we can understand the bagging effect in terms of a consensus of independent weak learners
>
> — Page 286, *The Elements of Statistical Learning*, 2016.

Both changes are made to seek less correlated trees but have the effect of training weaker, but perhaps not weak, models to comprise the ensemble.

- **Bagging**: explicitly trains weaker (but not weak) learners.

Consider stacked generalization (stacking, introduced in Chapter 28) that trains a model to best combine the predictions from multiple different models fit on the same training dataset. Each contributing level-0 model is in effect a strong learner, and the meta level-1 model seeks to make a stronger model by combining the predictions from the strong models.

- **Stacking**: explicitly combines the predictions from strong learners.

Mixture of experts (MoE) operates in a similar way, training multiple strong models (the experts) that are combined into hopefully stronger models via a meta-model, the gating network, and combing method.

> Mixture-of-experts can also be seen as a classifier selection algorithm, where individual classifiers are trained to become experts in some portion of the feature space. In this setting, individual classifiers are indeed trained to become experts, and hence are usually not weak classifiers
>
> — Page 16, *Ensemble Machine Learning*, 2012.

This highlights that although weak and strong learnability and learners are an important theoretical finding and basis for boosting, that the more generalized ideas of these classifiers are useful tools for designing and selecting ensemble methods.

20.5 Further Reading

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

Papers

- *The Strength of Weak Learnability*, 1990.
  https://link.springer.com/article/10.1007/BF00116037

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5
- *Ensemble Methods in Data Mining*, 2010.
  https://amzn.to/3frGM1A
- *The Elements of Statistical Learning*, 2016.
  https://amzn.to/31mzA31
- *Applied Predictive Modeling*, 2013.
  https://amzn.to/3eJyPVz

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning
- Boosting (machine learning), Wikipedia.
  https://en.wikipedia.org/wiki/Boosting_(machine_learning)

20.6 Summary

In this tutorial, you discovered weak and strong learners and their relationship with ensemble learning. Specifically, you learned:

- Weak learners are models that perform slightly better than random guessing.
- Strong learners are models that have arbitrarily good accuracy.
- Weak and strong learners are tools from computational learning theory and provide the basis for the development of the boosting class of ensemble methods.

Next

In the next section, we will take a closer look at the adaptive boosting ensemble algorithm.

Chapter 21

Adaptive Boost Ensemble

Boosting is a class of ensemble machine learning algorithms that involve combining the predictions from many weak learners. A weak learner is a model that is very simple, although has some skill on the dataset. Boosting was a theoretical concept long before a practical algorithm could be developed, and the AdaBoost (adaptive boosting) algorithm was the first successful approach for the idea. The AdaBoost algorithm involves using very short (one-level) decision trees as weak learners that are added sequentially to the ensemble. Each subsequent model attempts to correct the prediction errors made by the model before it in the sequence. This is achieved by weighing the training dataset to put more focus on training examples on which prior models made prediction errors. In this tutorial, you will discover how to develop AdaBoost ensembles for classification and regression. After completing this tutorial, you will know:

- AdaBoost ensemble is an ensemble created from decision trees added sequentially to the model
- How to use the AdaBoost ensemble for classification and regression with scikit-learn.
- How to explore the effect of AdaBoost model hyperparameters on model performance.

Let's get started.

21.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. AdaBoost Ensemble Algorithm
2. Evaluate AdaBoost Ensembles
3. AdaBoost Hyperparameters
4. Grid Search Hyperparameters

21.2 AdaBoost Ensemble Algorithm

Boosting refers to a class of machine learning ensemble algorithms where models are added sequentially and later models in the sequence correct the predictions made by earlier models in the sequence. AdaBoost, short for *Adaptive Boosting*, is a boosting ensemble machine learning algorithm, and was one of the first successful boosting approaches.

> We call the algorithm AdaBoost because, unlike previous algorithms, it adjusts adaptively to the errors of the weak hypotheses

— *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting*, 1996.

AdaBoost combines the predictions from short one-level decision trees, called decision stumps, although other algorithms can also be used. Decision stump algorithms are used as the AdaBoost algorithm seeks to use many weak models and correct their prediction errors by adding additional weak models. The training algorithm involves starting with one decision tree, finding those examples in the training dataset that were misclassified, and adding more weight to those examples. Another tree is trained on the same data, although now weighted by the misclassification errors. This process is repeated until a desired number of trees are added.

> If a training data point is misclassified, the weight of that training data point is increased (boosted). A second classifier is built using the new weights, which are no longer equal. Again, misclassified training data have their weights boosted and the procedure is repeated.

— *Multi-class AdaBoost*, 1996.

The algorithm was developed for classification and involves combining the predictions made by all decision trees in the ensemble. A similar approach was also developed for regression problems where predictions are made by using the average of the decision trees. The contribution of each model to the ensemble prediction is weighted based on the performance of the model on the training dataset.

> ... the new algorithm needs no prior knowledge of the accuracies of the weak hypotheses. Rather, it adapts to these accuracies and generates a weighted majority hypothesis in which the weight of each weak hypothesis is a function of its accuracy.

— *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting*, 1996.

Now that we are familiar with the AdaBoost algorithm, let's look at how we can fit AdaBoost models in Python.

21.3 Evaluate AdaBoost Ensembles

The scikit-learn Python machine learning library provides an implementation of AdaBoost for machine learning via the `AdaBoostRegressor` and `AdaBoostClassifier` classes. Both models operate the same way and take the same arguments that influence how the decision trees are created. Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model. When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions. Let's take a look at how to develop an AdaBoost ensemble for both classification and regression.

21.3.1 AdaBoost for Classification

In this section, we will look at using AdaBoost for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=6)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 21.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 21.2: Example output from creating the synthetic classification dataset.

Next, we can evaluate an AdaBoost algorithm on this dataset. We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate adaboost algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import AdaBoostClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=6)
# define the model
model = AdaBoostClassifier()
# define the evaluation method
```

```
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 21.3: Example of evaluating AdaBoost ensemble on the synthetic classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the AdaBoost ensemble with default hyperparameters achieves a classification accuracy of about 80 percent on this test dataset.

```
Mean Accuracy: 0.806 (0.041)
```

Listing 21.4: Example output from evaluating AdaBoost ensemble on the synthetic classification dataset.

We can also use the AdaBoost model as a final model and make predictions for classification. First, the AdaBoost ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make predictions using adaboost for classification
from sklearn.datasets import make_classification
from sklearn.ensemble import AdaBoostClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=6)
# define the model
model = AdaBoostClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [-3.47224758, 1.95378146, 0.04875169, -0.91592588, -3.54022468, 1.96405547,
    -7.72564954, -2.64787168, -1.81726906, -1.67104974, 2.33762043, -4.30273117, 0.4839841,
    -1.28253034, -10.6704077, -0.7641103, -3.58493721, 2.07283886, 0.08385173, 0.91461126]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 21.5: Example of making a prediction with an AdaBoost ensemble on the synthetic classification dataset.

Running the example fits the AdaBoost ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 0
```

Listing 21.6: Example output from making a prediction with an AdaBoost ensemble on the synthetic classification dataset.

Now that we are familiar with using AdaBoost for classification, let's look at the API for regression.

21.3.2 AdaBoost for Regression

In this section, we will look at using AdaBoost for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=6)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 21.7: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 21.8: Example output from creating the synthetic regression dataset.

Next, we can evaluate an AdaBoost algorithm on this dataset. As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate adaboost ensemble for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.ensemble import AdaBoostRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=6)
# define the model
model = AdaBoostRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 21.9: Example of evaluating AdaBoost ensemble on the synthetic regression dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the AdaBoost ensemble with default hyperparameters achieves a MAE of about 72.3.

```
MAE: -72.327 (4.041)
```

Listing 21.10: Example output from evaluating AdaBoost ensemble on the synthetic regression dataset.

We can also use the AdaBoost model as a final model and make predictions for regression. First, the AdaBoost ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# make predictions using adaboost for regression
from sklearn.datasets import make_regression
from sklearn.ensemble import AdaBoostRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=6)
# define the model
model = AdaBoostRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [1.20871625, 0.88440466, -0.9030013, -0.22687731, -0.82940077, -1.14410988,
    1.26554256, -0.2842871, 1.43929072, 0.74250241, 0.34035501, 0.45363034, 0.1778756,
    -1.75252881, -1.33337384, -1.50337215, -0.45099008, 0.46160133, 0.58385557, -1.79936198]
yhat = model.predict([row])
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Listing 21.11: Example of making a prediction with an AdaBoost ensemble on the synthetic regression dataset.

Running the example fits the AdaBoost ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: -10
```

Listing 21.12: Example output from making a prediction with an AdaBoost ensemble on the synthetic regression dataset.

Now that we are familiar with using the scikit-learn API to evaluate and use AdaBoost ensembles, let's look at configuring the model.

21.4 AdaBoost Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the AdaBoost ensemble and their effect on model performance.

21.4.1 Explore Number of Trees

An important hyperparameter for the AdaBoost algorithm is the number of decision trees used in the ensemble. Recall that each decision tree used in the ensemble is designed to be a weak learner. That is, it has skill over random prediction, but is not highly skillful. As such, one-level decision trees are used, called decision stumps. The number of trees added to the model must be high for the model to work well, often hundreds, if not thousands. The number of trees can be set via the `n_estimators` argument and defaults to 50. The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore adaboost ensemble number of trees effect on performance
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.ensemble import AdaBoostClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=6)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define number of trees to consider
  n_trees = [10, 50, 100, 500, 1000, 5000]
  for n in n_trees:
    models[str(n)] = AdaBoostClassifier(n_estimators=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
```

```
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 21.13: Example of evaluating the effect of varying the number of trees for the AdaBoost ensemble.

Running the example first reports the mean accuracy for each configured number of decision trees.

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

In this case, we can see that performance improves on this dataset until about 50 trees and declines after that. This might be a sign of the ensemble overfitting the training dataset after additional trees are added.

```
>10 0.773 (0.039)
>50 0.806 (0.041)
>100 0.801 (0.032)
>500 0.793 (0.028)
>1000 0.791 (0.032)
>5000 0.782 (0.031)
```

Listing 21.14: Example output from evaluating the effect of varying the number of trees for the AdaBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees. We can see the general trend of model performance and ensemble size.

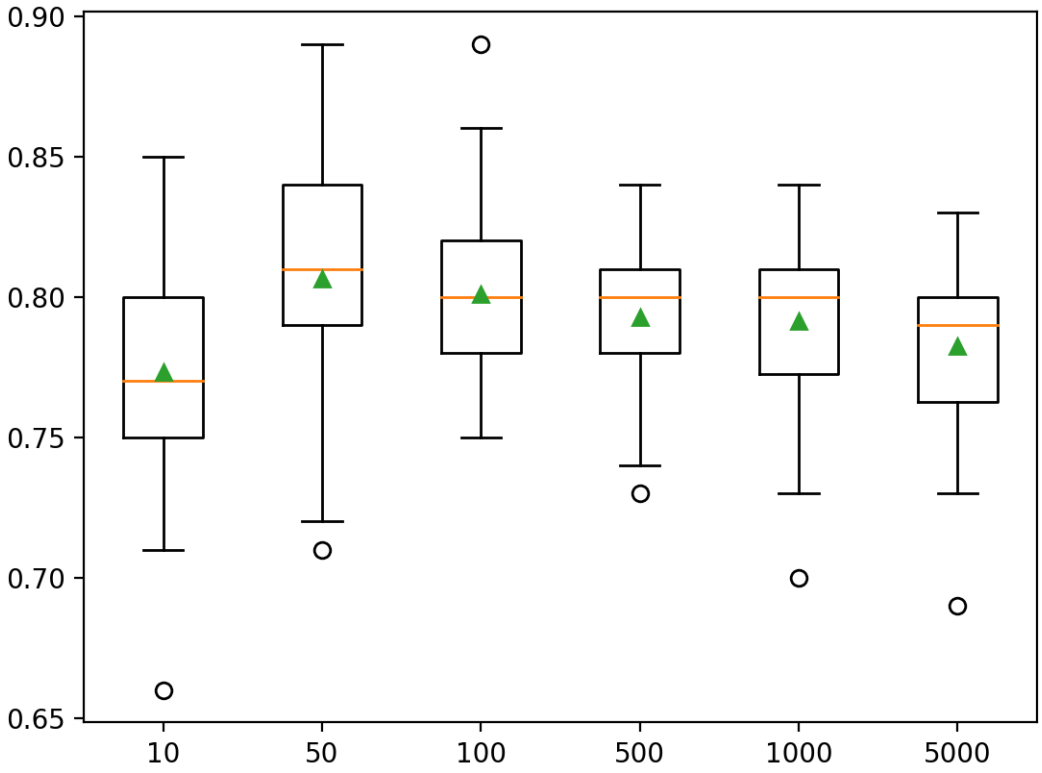

Figure 21.1: Box Plot of AdaBoost Ensemble Size vs. Classification Accuracy.

21.4.2 Explore Weak Learner

A decision tree with one level is used as the weak learner by default. We can make the models used in the ensemble less weak (more skillful) by increasing the depth of the decision tree. This can be achieved via the `max_depth` argument on the `DecisionTreeClassifier` class used as the base estimator. The default is value is 1 (e.g. a decision stump) when the decision tree is used as the base learner, although we will explore values from 1 to 10. The example below explores the effect of increasing the depth of the `DecisionTreeClassifier` weak learner on the AdBoost ensemble.

```
# explore adaboost ensemble tree depth effect on performance
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.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
```

```
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=6)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore depths from 1 to 10
  for i in range(1,11):
    # define base model
    base = DecisionTreeClassifier(max_depth=i)
    # define ensemble model
    models[str(i)] = AdaBoostClassifier(base_estimator=base)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 21.15: Example of evaluating the effect of varying the tree depth for the AdaBoost ensemble.

Running the example first reports the mean accuracy for each configured weak learner tree depth.

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

In this case, we can see that as the depth of the decision trees is increased, the performance of the ensemble is also increased on this dataset.

```
>1 0.806 (0.041)
```

```
>2 0.864 (0.028)
>3 0.867 (0.030)
>4 0.889 (0.029)
>5 0.909 (0.021)
>6 0.923 (0.020)
>7 0.927 (0.025)
>8 0.928 (0.028)
>9 0.923 (0.017)
>10 0.926 (0.030)
```

Listing 21.16: Example output from evaluating the effect of varying the tree depth for the AdaBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured tree depth. We can see the general trend of increased model performance with tree depth.

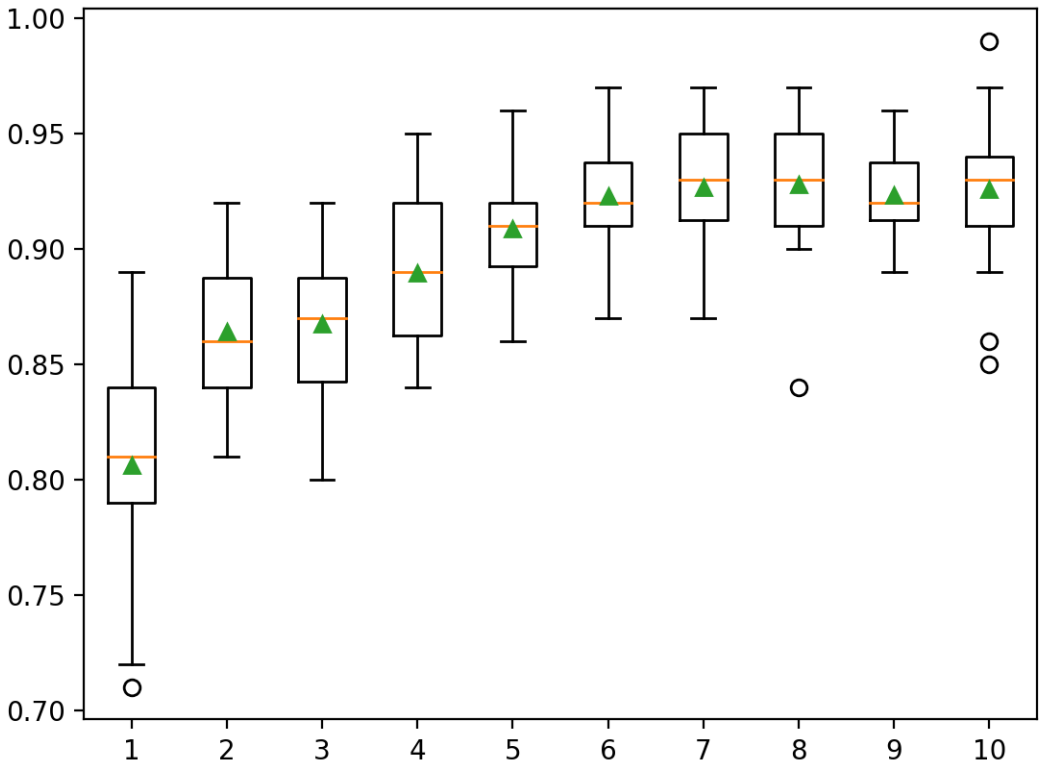

Figure 21.2: Box Plot of AdaBoost Ensemble Weak Learner Depth vs. Classification Accuracy.

21.4.3 Explore Learning Rate

AdaBoost also supports a learning rate (sometimes called shrinkage) that controls the contribution of each model to the ensemble prediction. This is controlled by the `learning_rate` argument and by default is set to 1.0 or full contribution. Smaller or larger values might be appropriate depending on the number of models used in the ensemble. There is a balance

between the contribution of the models and the number of trees in the ensemble. More trees may require a smaller learning rate; fewer trees may require a larger learning rate. It is common to use values between 0 and 1 and sometimes very small values to avoid overfitting such as 0.1, 0.01 or 0.001. The example below explores learning rate values between 0.1 and 2.0 in 0.1 increments.

```
# explore adaboost ensemble learning rate effect on performance
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import AdaBoostClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=6)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore learning rates from 0.1 to 2 in 0.1 increments
  for i in arange(0.1, 2.1, 0.1):
    key = '%.1f' % i
    models[key] = AdaBoostClassifier(learning_rate=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
```

```
pyplot.xticks(rotation=45)
pyplot.show()
```

Listing 21.17: Example of evaluating the effect of varying the learning rate for the AdaBoost ensemble.

Running the example first reports the mean accuracy for each configured learning rate.

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

In this case, we can see similar values between 0.5 to 1.0 and a decrease in model performance after that.

```
>0.1 0.767 (0.049)
>0.2 0.786 (0.042)
>0.3 0.802 (0.040)
>0.4 0.798 (0.037)
>0.5 0.805 (0.042)
>0.6 0.795 (0.031)
>0.7 0.799 (0.035)
>0.8 0.801 (0.033)
>0.9 0.805 (0.032)
>1.0 0.806 (0.041)
>1.1 0.801 (0.037)
>1.2 0.800 (0.030)
>1.3 0.799 (0.041)
>1.4 0.793 (0.041)
>1.5 0.790 (0.040)
>1.6 0.775 (0.034)
>1.7 0.767 (0.054)
>1.8 0.768 (0.040)
>1.9 0.736 (0.047)
>2.0 0.682 (0.048)
```

Listing 21.18: Example output from evaluating the effect of varying the learning rate for the AdaBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured learning rate. We can see the general trend of decreasing model performance with a learning rate larger than 1.0 on this dataset.

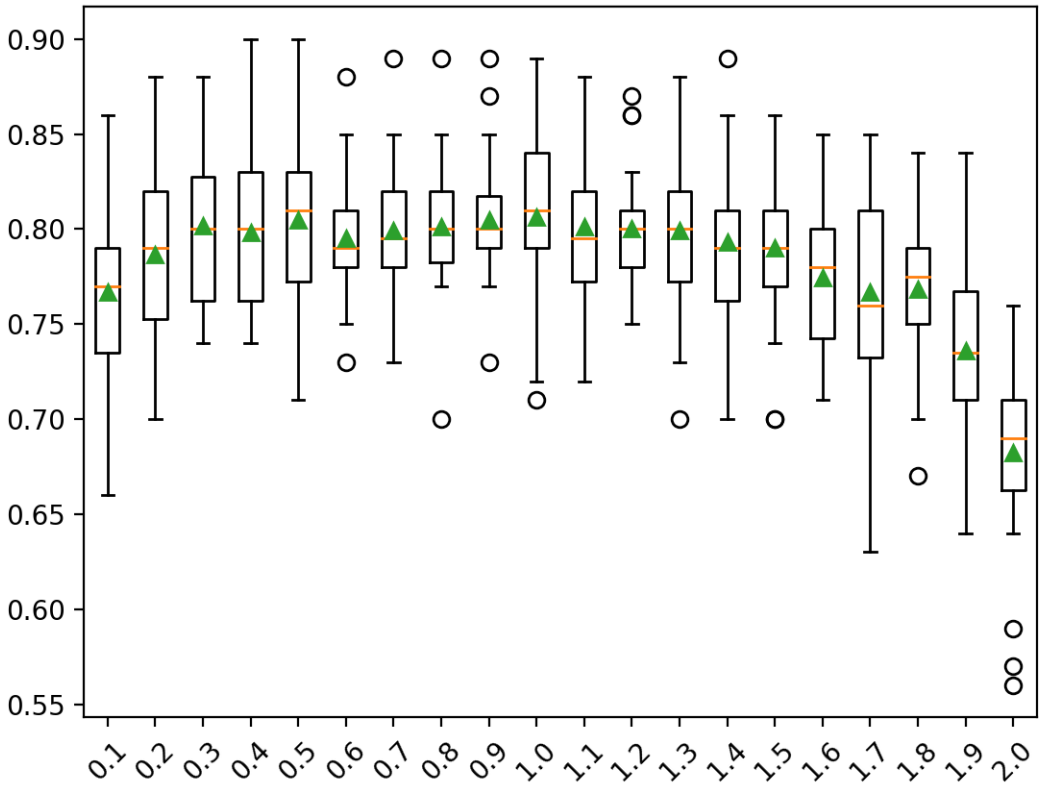

Figure 21.3: Box Plot of AdaBoost Ensemble Learning Rate vs. Classification Accuracy.

21.4.4 Explore Alternate Algorithm

The default algorithm used in the ensemble is a decision tree, although other algorithms can be used. The intent is to use very simple models, called weak learners. Also, the scikit-learn implementation requires that any models used must also support weighted samples, as they are how the ensemble is created by fitting models based on a weighted version of the training dataset. The base-model can be specified via the `base_estimator` argument. The base-model must also support predicting probabilities or probability-like scores in the case of classification. If the specified model does not support a weighted training dataset, you will see an error message as follows:

```
ValueError: KNeighborsClassifier doesn't support sample_weight.
```

Listing 21.19: Example output error from using an unsupported base estimator.

One example of a model that supports a weighted training is the logistic regression algorithm. The example below demonstrates an AdaBoost algorithm with a `LogisticRegression` weak learner.

```
# evaluate adaboost algorithm with logistic regression weak learner for classification
from numpy import mean
from numpy import std
```

```
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import AdaBoostClassifier
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=6)
# define the model
model = AdaBoostClassifier(base_estimator=LogisticRegression())
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 21.20: Example of evaluating the effect of changing the base estimator for the AdaBoost ensemble.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the AdaBoost ensemble with a logistic regression weak model achieves a classification accuracy of about 79 percent on this test dataset.

```
Mean Accuracy: 0.794 (0.032)
```

Listing 21.21: Example output from evaluating the effect of changing the base estimator for the AdaBoost ensemble.

21.5 Grid Search Hyperparameters

AdaBoost can be challenging to configure as the algorithm has many key hyperparameters that influence the behavior of the model on training data and the hyperparameters interact with each other. As such, it is a good practice to use a search process to discover a configuration of the model hyperparameters that works well or best for a given predictive modeling problem. Popular search processes include a random search and a grid search. In this section we will look at grid searching common ranges for the key hyperparameters for the AdaBoost algorithm that you can use as starting point for your own projects. This can be achieved using the `GridSearchCV` class and specifying a dictionary that maps model hyperparameter names to the values to search.

In this case, we will grid search two key hyperparameters for AdaBoost: the number of trees used in the ensemble and the learning rate. We will use a range of popular well performing values for each hyperparameter. Each configuration combination will be evaluated using repeated k-fold cross-validation and configurations will be compared using the mean score, in this case, classification accuracy. The complete example of grid searching the key hyperparameters of the AdaBoost algorithm on our synthetic classification dataset is listed below.

```
# example of grid searching key hyperparameters for adaboost on a classification dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import AdaBoostClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=6)
# define the model with default hyperparameters
model = AdaBoostClassifier()
# define the grid of values to search
grid = dict()
grid['n_estimators'] = [10, 50, 100, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
    scoring='accuracy')
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 21.22: Example of a grid search of key hyperparameters for the AdaBoost algorithm.

Running the example many take a while depending on your hardware. At the end of the run, the configuration that achieved the best score is reported first, followed by the scores for all other configurations that were considered.

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

In this case, we can see that a configuration with 500 trees and a learning rate of 0.1 performed the best with a classification accuracy of about 81.3 percent. The model may perform even better with more trees such as 1,000 or 5,000 although these configurations were not tested in this case to ensure that the grid search completed in a reasonable time.

```
Best: 0.813667 using {'learning_rate': 0.1, 'n_estimators': 500}
0.646333 (0.036376) with: {'learning_rate': 0.0001, 'n_estimators': 10}
0.646667 (0.036545) with: {'learning_rate': 0.0001, 'n_estimators': 50}
0.646667 (0.036545) with: {'learning_rate': 0.0001, 'n_estimators': 100}
0.647000 (0.038136) with: {'learning_rate': 0.0001, 'n_estimators': 500}
0.646667 (0.036545) with: {'learning_rate': 0.001, 'n_estimators': 10}
0.647000 (0.038136) with: {'learning_rate': 0.001, 'n_estimators': 50}
0.654333 (0.045511) with: {'learning_rate': 0.001, 'n_estimators': 100}
0.672667 (0.046543) with: {'learning_rate': 0.001, 'n_estimators': 500}
```

```
0.648333 (0.042197) with: {'learning_rate': 0.01, 'n_estimators': 10}
0.671667 (0.045613) with: {'learning_rate': 0.01, 'n_estimators': 50}
0.715000 (0.053213) with: {'learning_rate': 0.01, 'n_estimators': 100}
0.767667 (0.045948) with: {'learning_rate': 0.01, 'n_estimators': 500}
0.716667 (0.048876) with: {'learning_rate': 0.1, 'n_estimators': 10}
0.767000 (0.049271) with: {'learning_rate': 0.1, 'n_estimators': 50}
0.784667 (0.042874) with: {'learning_rate': 0.1, 'n_estimators': 100}
0.813667 (0.032092) with: {'learning_rate': 0.1, 'n_estimators': 500}
0.773333 (0.038759) with: {'learning_rate': 1.0, 'n_estimators': 10}
0.806333 (0.040701) with: {'learning_rate': 1.0, 'n_estimators': 50}
0.801000 (0.032491) with: {'learning_rate': 1.0, 'n_estimators': 100}
0.792667 (0.027560) with: {'learning_rate': 1.0, 'n_estimators': 500}
```

Listing 21.23: Example output from a grid search of key hyperparameters for the AdaBoost algorithm.

21.6 Further Reading

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

Papers

- *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting*, 1996.
  https://link.springer.com/chapter/10.1007/3-540-59119-2_166
- *Multi-class AdaBoost*, 2009.
  https://www.intlpress.com/site/pub/pages/journals/items/sii/content/vols/0002/0003/a008/
- *Improving Regressors using Boosting Techniques*, 1997.
  https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.5683&rank=1

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

APIs

- `sklearn.ensemble.AdaBoostRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html

- `sklearn.ensemble.AdaBoostClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

Articles

- Boosting (machine learning), Wikipedia.
  https://en.wikipedia.org/wiki/Boosting_(machine_learning)
- AdaBoost, Wikipedia.
  https://en.wikipedia.org/wiki/AdaBoost

21.7 Summary

In this tutorial, you discovered how to develop AdaBoost ensembles for classification and regression. Specifically, you learned:

- AdaBoost ensemble is an ensemble created from decision trees added sequentially to the model.
- How to use the AdaBoost ensemble for classification and regression with scikit-learn.
- How to explore the effect of AdaBoost model hyperparameters on model performance.

Next

In the next section, we will take a closer look at the gradient boosting ensemble algorithm.

Chapter 22

Gradient Boosting Ensemble

The Gradient Boosting Machine is a powerful ensemble machine learning algorithm that uses decision trees. Boosting is a general ensemble technique that involves sequentially adding models to the ensemble where subsequent models correct the performance of prior models. AdaBoost was the first algorithm to deliver on the promise of boosting. Gradient boosting is a generalization of AdaBoosting, improving the performance of the approach and introducing ideas from bootstrap aggregation to further improve the models, such as randomly sampling the samples and features when fitting ensemble members.

Gradient boosting performs well, if not the best, on a wide range of tabular datasets, and versions of the algorithm like XGBoost and LightBoost often play an important role in winning machine learning competitions. In this tutorial, you will discover how to develop Gradient Boosting ensembles for classification and regression. After completing this tutorial, you will know:

- Gradient Boosting ensemble is an ensemble created from decision trees added sequentially to the model.
- How to use the Gradient Boosting ensemble for classification and regression with scikit-learn.
- How to explore the effect of Gradient Boosting model hyperparameters on model performance.

Let's get started.

22.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Gradient Boosting Algorithm
2. Evaluate Gradient Boosting Ensembles
3. Gradient Boosting Hyperparameters
4. Grid Search Hyperparameters
5. Common Questions

22.2 Gradient Boosting Machines Algorithm

Gradient boosting refers to a class of ensemble machine learning algorithms that can be used for classification or regression predictive modeling problems. Gradient boosting is also known as gradient tree boosting, stochastic gradient boosting (an extension), and gradient boosting machines, or GBM for short. Ensembles are constructed from decision tree models. Trees are added one at a time to the ensemble and fit to correct the prediction errors made by prior models. This is a type of ensemble machine learning model referred to as boosting.

Models are fit using any arbitrary differentiable loss function and gradient descent optimization algorithm. This gives the technique its name, *gradient boosting*, as the loss gradient is minimized as the model is fit, much like a neural network.

> One way to produce a weighted combination of classifiers which optimizes [the cost] is by gradient descent in function space
>
> — *Boosting Algorithms as Gradient Descent in Function Space*, 1999.

Naive gradient boosting is a greedy algorithm and can overfit the training dataset quickly. It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting. There are three types of enhancements to basic gradient boosting that can improve performance:

- **Tree Constraints**: such as the depth of the trees and the number of trees used in the ensemble.
- **Weighted Updates**: such as a learning rate used to limit how much each tree contributes to the ensemble.
- **Random Sampling**: such as fitting trees on random subsets of features and samples.

The use of random sampling often leads to a change in the name of the algorithm to *stochastic gradient boosting*.

> ... at each iteration a subsample of the training data is drawn at random (without replacement) from the full training dataset. The randomly selected subsample is then used, instead of the full sample, to fit the base learner.
>
> — *Stochastic Gradient Boosting*, 1999.

Gradient boosting is an effective machine learning algorithm and is often the main, or one of the main, algorithms used to win machine learning competitions (like Kaggle) on tabular and similar structured datasets. Now that we are familiar with the gradient boosting algorithm, let's look at how we can fit GBM models in Python.

22.3 Evaluate Gradient Boosting Ensembles

The scikit-learn Python machine learning library provides an implementation of AdaBoost for machine learning via the `GradientBoostingRegressor` and `GradientBoostingClassifier` classes. Both models operate the same way and take the same arguments that influence how the decision trees are created and added to the ensemble. Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model. When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions. Let's take a look at how to develop a Gradient Boosting ensemble for both classification and regression.

22.3.1 Gradient Boosting for Classification

In this section, we will look at using Gradient Boosting for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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 22.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 22.2: Example output from creating the synthetic classification dataset.

Next, we can evaluate a Gradient Boosting algorithm on this dataset. We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate gradient boosting algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = GradientBoostingClassifier()
```

```
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 22.3: Example of evaluating a gradient boosting ensemble the synthetic classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Gradient Boosting ensemble with default hyperparameters achieves a classification accuracy of about 89.9 percent on this test dataset.

```
Mean Accuracy: 0.899 (0.030)
```

Listing 22.4: Example output from evaluating a gradient boosting ensemble the synthetic classification dataset.

We can also use the Gradient Boosting model as a final model and make predictions for classification. First, the Gradient Boosting ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make predictions using gradient boosting for classification
from sklearn.datasets import make_classification
from sklearn.ensemble import GradientBoostingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = GradientBoostingClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719, 0.28422388,
    -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799, 3.34692332,
    4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 22.5: Example of making a prediction with a gradient boosting ensemble for classification.

Running the example fits the Gradient Boosting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 22.6: Example output from making a prediction with a gradient boosting ensemble for classification.

Now that we are familiar with using Gradient Boosting for classification, let's look at the API for regression.

22.3.2 Gradient Boosting for Regression

In this section, we will look at using Gradient Boosting for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 22.7: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 22.8: Example output from creating the synthetic regression dataset.

Next, we can evaluate a Gradient Boosting algorithm on this dataset. As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate gradient boosting ensemble for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.ensemble import GradientBoostingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# define the model
model = GradientBoostingRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 22.9: Example of evaluating a gradient boosting ensemble the synthetic regression dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the Gradient Boosting ensemble with default hyperparameters achieves a MAE of about 62.

```
MAE: -62.475 (3.254)
```

Listing 22.10: Example output from evaluating a gradient boosting ensemble the synthetic regression dataset.

We can also use the Gradient Boosting model as a final model and make predictions for regression. First, the Gradient Boosting ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# gradient boosting ensemble for making predictions for regression
from sklearn.datasets import make_regression
from sklearn.ensemble import GradientBoostingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# define the model
model = GradientBoostingRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.20543991, -0.97049844, -0.81403429, -0.23842689, -0.60704084, -0.48541492,
    0.53113006, 2.01834338, -0.90745243, -1.85859731, -1.02334791, -0.6877744, 0.60984819,
    -0.70630121, -1.29161497, 1.32385441, 1.42150747, 1.26567231, 2.56569098, -0.11154792]
yhat = model.predict([row])
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Listing 22.11: Example of making a prediction with a gradient boosting ensemble for regression.

Running the example fits the Gradient Boosting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: 37
```

Listing 22.12: Example output from making a prediction with a gradient boosting ensemble for regression.

Now that we are familiar with using the scikit-learn API to evaluate and use Gradient Boosting ensembles, let's look at configuring the model.

22.4 Gradient Boosting Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the Gradient Boosting ensemble and their effect on model performance. There are many hyperparameters that could be tuned for gradient boosting. It may be best practice to use a search procedure such as random search or grid search to discover the combination of hyperparameters that works best for a given dataset.

There are perhaps four key hyperparameters that have the biggest effect on model performance, they are the number of models in the ensemble, the learning rate, the variance of the model controlled via the size of the data sample used to train each model or features used in tree splits, and finally the depth of the decision tree. We will take a closer look at the effect each of these hyperparameters in isolation in this section, although they all interact and should be tuned together or in pairs, such as learning rate with ensemble size, and sample size/number of features with tree depth.

22.4.1 Explore Number of Trees

An important hyperparameter for the Gradient Boosting ensemble algorithm is the number of decision trees used in the ensemble. Recall that decision trees are added to the model sequentially in an effort to correct and improve upon the predictions made by prior trees. As such, more trees is often better. The number of trees must also be balanced with the learning rate, e.g. more trees may require a smaller learning rate, fewer trees may require a larger learning rate. The number of trees can be set via the `n_estimators` argument and defaults to 100. The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore gradient boosting number of trees effect on performance
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.ensemble import GradientBoostingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define number of trees to consider
  n_trees = [10, 50, 100, 500, 1000, 5000]
  for n in n_trees:
    models[str(n)] = GradientBoostingClassifier(n_estimators=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
```

```
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 22.13: Example of evaluating the effect of varying the number of trees on the gradient boosting ensemble.

Running the example first reports the mean accuracy for each configured number of decision trees.

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

In this case, we can see that performance improves on this dataset until about 500 trees, after which performance appears to level off. Unlike AdaBoost, Gradient Boosting appears to not overfit as the number of trees is increased in this case.

```
>10 0.830 (0.037)
>50 0.880 (0.033)
>100 0.899 (0.030)
>500 0.919 (0.025)
>1000 0.919 (0.025)
>5000 0.918 (0.026)
```

Listing 22.14: Example output from evaluating the effect of varying the number of trees on the gradient boosting ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees. We can see the general trend of increasing model performance and ensemble size.

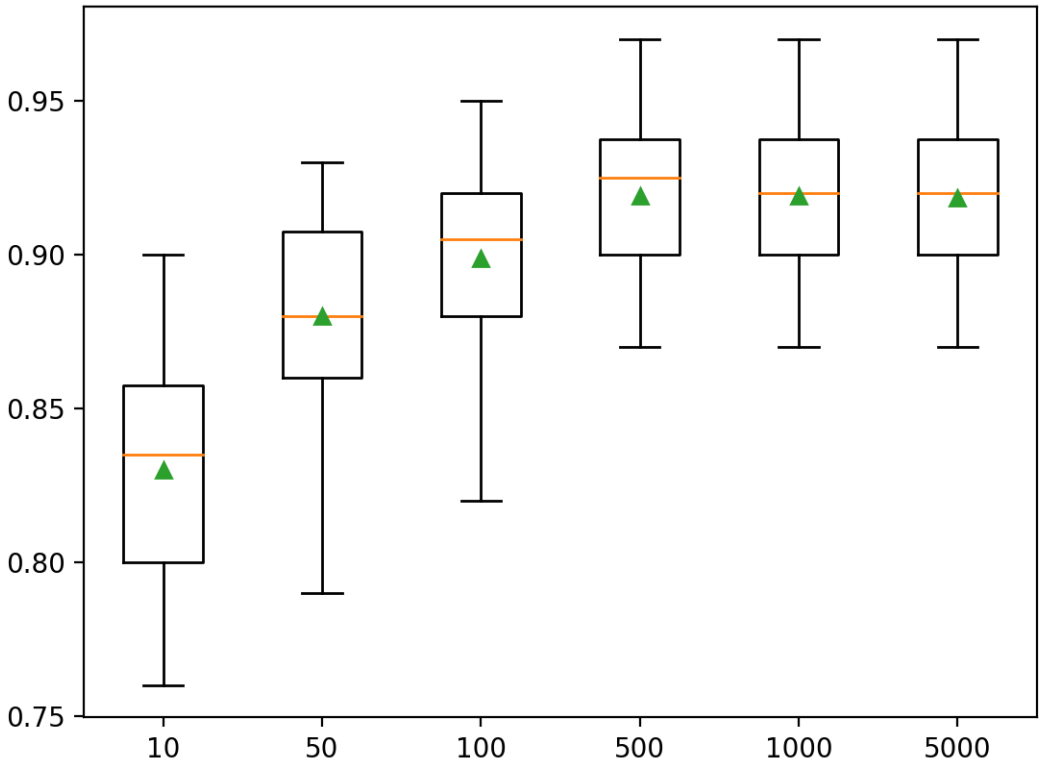

Figure 22.1: Box Plot of Gradient Boosting Ensemble Size vs. Classification Accuracy.

22.4.2 Explore Number of Samples

The number of samples used to fit each tree can be varied. This means that each tree is fit on a randomly selected subset of the training dataset. Using fewer samples introduces more variance for each tree, although it can improve the overall performance of the model. The number of samples used to fit each tree is specified by the `subsample` argument and can be set to a fraction of the training dataset size. By default, it is set to 1.0 to use the entire training dataset. The example below demonstrates the effect of the sample size on model performance.

```
# explore gradient boosting ensemble number of samples effect on performance
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
```

```
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore sample ratio from 10% to 100% in 10% increments
  for i in arange(0.1, 1.1, 0.1):
    key = '%.1f' % i
    models[key] = GradientBoostingClassifier(subsample=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 22.15: Example of evaluating the effect of varying the number of samples on the gradient boosting ensemble.

Running the example first reports the mean accuracy for each configured sample size.

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

In this case, we can see that mean performance is probably best for a sample size that is about half the size of the training dataset, such as 0.4 or higher.

```
>0.1 0.876 (0.044)
>0.2 0.897 (0.029)
>0.3 0.897 (0.032)
>0.4 0.900 (0.031)
>0.5 0.910 (0.030)
>0.6 0.904 (0.029)
```

```
>0.7 0.909 (0.027)
>0.8 0.900 (0.031)
>0.9 0.901 (0.029)
>1.0 0.899 (0.030)
```

Listing 22.16: Example output from evaluating the effect of varying the number of samples on the gradient boosting ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured sample size. We can see the general trend of increasing model performance perhaps peaking around 0.4 and staying somewhat level.

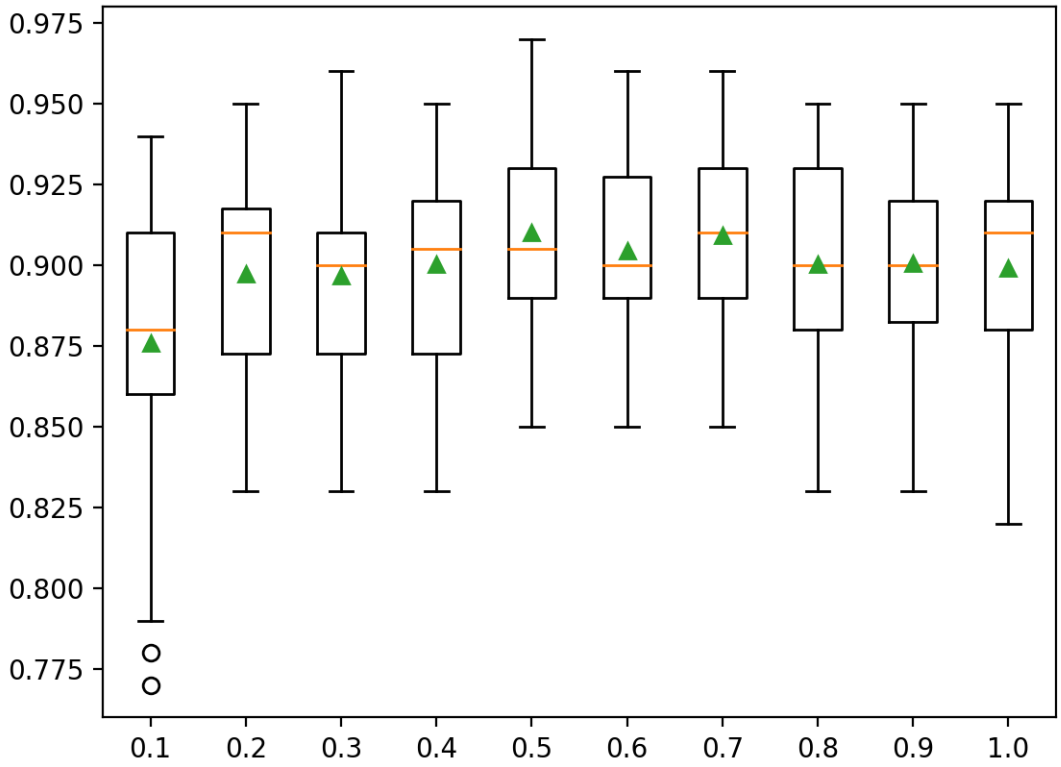

Figure 22.2: Box Plot of Gradient Boosting Ensemble Sample Size vs. Classification Accuracy.

22.4.3 Explore Number of Features

The number of features used to fit each decision tree can be varied. Like changing the number of samples, changing the number of features introduces additional variance into the model, which may improve performance, although it might require an increase in the number of trees. The number of features used by each tree is taken as a random sample and is specified by the `max_features` argument and defaults to all features in the training dataset. The example below explores the effect of the number of features on model performance for the test dataset between 1 and 20.

```
# explore gradient boosting number of features on performance
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.ensemble import GradientBoostingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore number of features from 1 to 20
  for i in range(1,21):
    models[str(i)] = GradientBoostingClassifier(max_features=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 22.17: Example of evaluating the effect of varying the number of features on the gradient boosting ensemble.

Running the example first reports the mean accuracy for each configured number of features.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation

procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that mean performance increases to about half the number of features and stays somewhat level after that. It's surprising that removing half of the input variables has so little effect.

```
>1 0.864 (0.036)
>2 0.885 (0.032)
>3 0.891 (0.031)
>4 0.893 (0.036)
>5 0.898 (0.030)
>6 0.898 (0.032)
>7 0.892 (0.032)
>8 0.901 (0.032)
>9 0.900 (0.029)
>10 0.895 (0.034)
>11 0.899 (0.032)
>12 0.899 (0.030)
>13 0.898 (0.029)
>14 0.900 (0.033)
>15 0.901 (0.032)
>16 0.897 (0.028)
>17 0.902 (0.034)
>18 0.899 (0.032)
>19 0.899 (0.032)
>20 0.899 (0.030)
```

Listing 22.18: Example output from evaluating the effect of varying the number of features on the gradient boosting ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of features. We can see the general trend of increasing model performance perhaps peaking around eight or nine features and staying somewhat level.

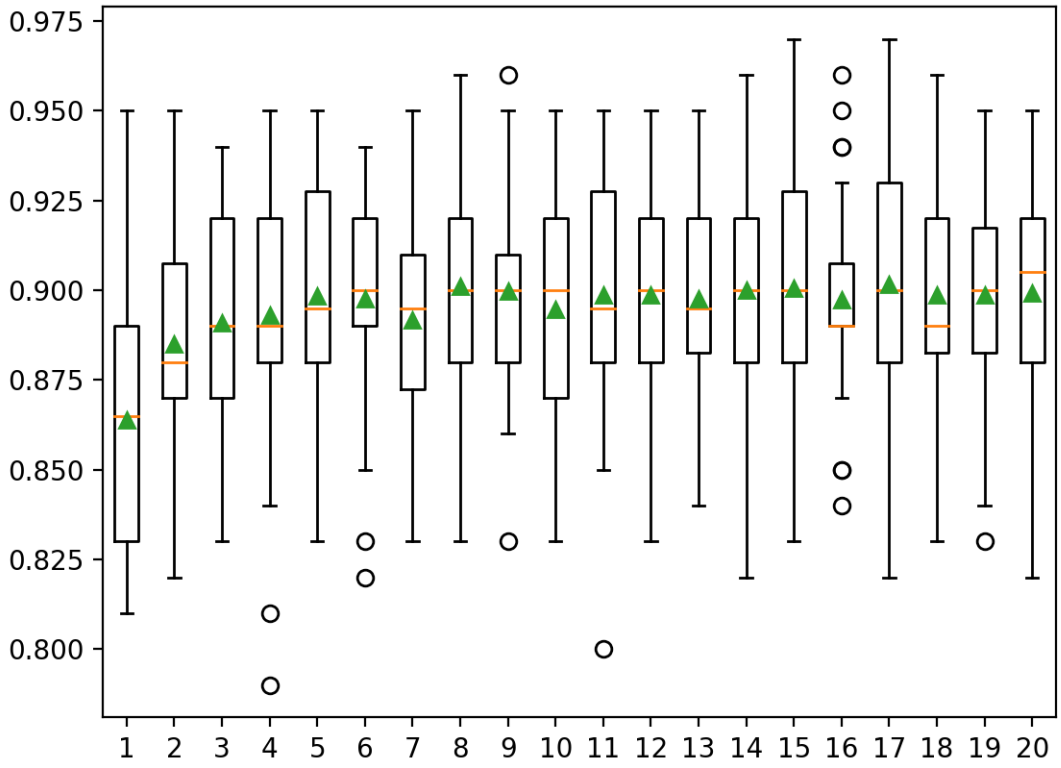

Figure 22.3: Box Plot of Gradient Boosting Ensemble Number of Features vs. Classification Accuracy.

22.4.4 Explore Learning Rate

Learning rate, also called shrinkage controls the amount of contribution that each model has on the ensemble prediction. Smaller rates may require more decision trees in the ensemble, whereas larger rates may require an ensemble with fewer trees. It is common to explore learning rate values on a log scale, such as between a very small value like 0.0001 and 1.0. The learning rate can be controlled via the `learning_rate` argument and defaults to 0.1. The example below explores the learning rate and compares the effect of values between 0.0001 and 1.0.

```
# explore gradient boosting ensemble learning rate effect on performance
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.ensemble import GradientBoostingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
```

```
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define learning rates to explore
  for i in [0.0001, 0.001, 0.01, 0.1, 1.0]:
    key = '%.4f' % i
    models[key] = GradientBoostingClassifier(learning_rate=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 22.19: Example of evaluating the effect of varying the learning rate on the gradient boosting ensemble.

Running the example first reports the mean accuracy for each configured learning rate.

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

In this case, we can see that a larger learning rate results in better performance on this dataset. We would expect that adding more trees to the ensemble for the smaller learning rates would further lift performance. This highlights the trade-off between the number of trees (speed of training) and learning rate, e.g. we can fit a model faster by using fewer trees and a larger learning rate at the cost of overfitting.

```
>0.0001 0.761 (0.043)
>0.0010 0.781 (0.034)
```

```
>0.0100 0.836 (0.034)
>0.1000 0.898 (0.030)
>1.0000 0.907 (0.028)
```

Listing 22.20: Example output from evaluating the effect of varying the learning rate on the gradient boosting ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured learning rate. We can see the general trend of increasing model performance with the increase in learning rate.

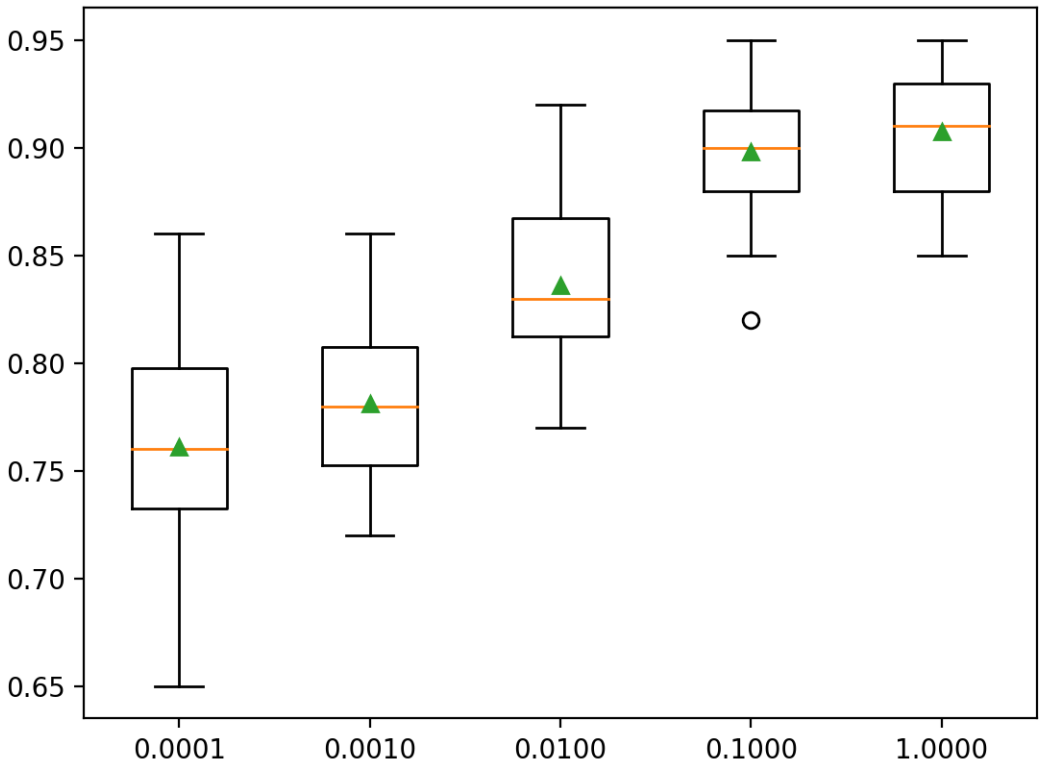

Figure 22.4: Box Plot of Gradient Boosting Ensemble Learning Rate vs. Classification Accuracy.

22.4.5 Explore Tree Depth

Like varying the number of samples and features used to fit each decision tree, varying the depth of each tree is another important hyperparameter for gradient boosting. The tree depth controls how specialized each tree is to the training dataset: how general or overfit it might be. Trees are preferred that are not too shallow and general (like AdaBoost) and not too deep and specialized (like bootstrap aggregation). Gradient boosting performs well with trees that have a modest depth finding a balance between skill and generality. Tree depth is controlled via the `max_depth` argument and defaults to 3. The example below explores tree depths between 1 and 10 and the effect on model performance.

```
# explore gradient boosting tree depth effect on performance
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.ensemble import GradientBoostingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define max tree depths to explore between 1 and 10
  for i in range(1,11):
    models[str(i)] = GradientBoostingClassifier(max_depth=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 22.21: Example of evaluating the effect of varying the tree depth on the gradient boosting ensemble.

Running the example first reports the mean accuracy for each configured tree depth.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation

procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that performance improves with tree depth, perhaps peaking around a depth of 3 to 6, after which the deeper, more specialized trees result in worse performance.

```
>1 0.834 (0.031)
>2 0.877 (0.029)
>3 0.899 (0.030)
>4 0.905 (0.032)
>5 0.916 (0.030)
>6 0.912 (0.031)
>7 0.908 (0.033)
>8 0.888 (0.031)
>9 0.853 (0.036)
>10 0.835 (0.034)
```

Listing 22.22: Example output from evaluating the effect of varying the three depth on the gradient boosting ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured tree depth. We can see the general trend of increasing model performance with the tree depth to a point, after which performance begins to degrade rapidly with the over-specialized trees.

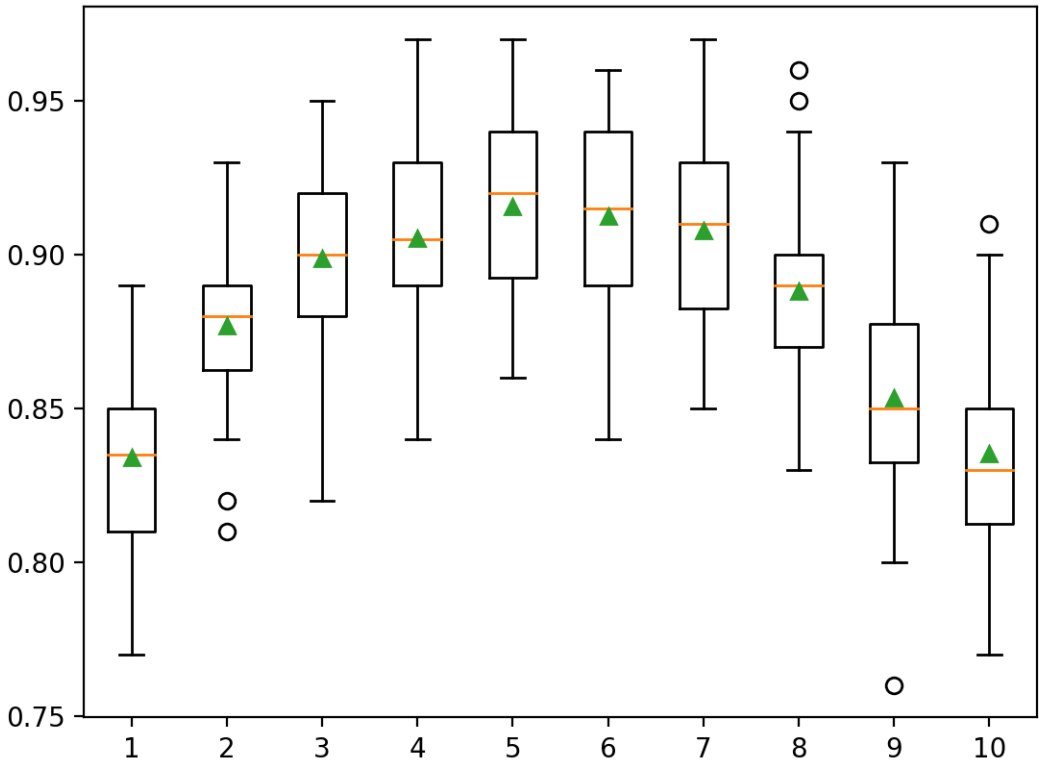

Figure 22.5: Box Plot of Gradient Boosting Ensemble Tree Depth vs. Classification Accuracy.

22.5 Grid Search Hyperparameters

Gradient boosting can be challenging to configure as the algorithm has many key hyperparameters that influence the behavior of the model on training data and the hyperparameters interact with each other. As such, it is a good practice to use a search process to discover a configuration of the model hyperparameters that works well or best for a given predictive modeling problem. Popular search processes include a random search and a grid search. In this section we will look at grid searching common ranges for the key hyperparameters for the gradient boosting algorithm that you can use as starting point for your own projects. This can be achieving using the `GridSearchCV` class and specifying a dictionary that maps model hyperparameter names to the values to search.

In this case, we will grid search four key hyperparameters for gradient boosting: the number of trees used in the ensemble, the learning rate, subsample size used to train each tree, and the maximum depth of each tree. We will use a range of popular well performing values for each hyperparameter. Each configuration combination will be evaluated using repeated k-fold cross-validation and configurations will be compared using the mean score, in this case, classification accuracy. The complete example of grid searching the key hyperparameters of the gradient boosting algorithm on our synthetic classification dataset is listed below.

```
# example of grid searching key hyperparameters for gradient boosting
from sklearn.datasets import make_classification
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import GradientBoostingClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model with default hyperparameters
model = GradientBoostingClassifier()
# define the grid of values to search
grid = dict()
grid['n_estimators'] = [10, 50, 100, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['subsample'] = [0.5, 0.7, 1.0]
grid['max_depth'] = [3, 7, 9]
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
    scoring='accuracy')
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 22.23: Example of a grid search of key hyperparameters for the Gradient Boosting algorithm.

Running the example many take a while depending on your hardware. At the end of the run, the configuration that achieved the best score is reported first, followed by the scores for all other configurations that were considered.

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

In this case, we can see that a configuration with a learning rate of 0.1, max depth of 7 levels, 500 trees and a subsample of 70% performed the best with a classification accuracy of about 94.6 percent. The model may perform even better with more trees such as 1,000 or 5,000 although these configurations were not tested in this case to ensure that the grid search completed in a reasonable time.

```
Best: 0.946667 using {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 500,
    'subsample': 0.7}
0.529667 (0.089012) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 10,
    'subsample': 0.5}
0.525667 (0.077875) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 10,
    'subsample': 0.7}
0.524000 (0.072874) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 10,
    'subsample': 1.0}
0.772667 (0.037500) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 50,
    'subsample': 0.5}
0.767000 (0.037696) with: {'learning_rate': 0.0001, 'max_depth': 3, 'n_estimators': 50,
    'subsample': 0.7}
...
```

Listing 22.24: Example output from a grid search of key hyperparameters for the Gradient Boosting algorithm.

22.6 Common Questions

In this section we will take a closer look at some common sticking points you may have with the gradient boosting ensemble procedure.

Q. What algorithm should be used in the ensemble?

Technically any high variance algorithm that support instance weighting can be used as the basis for the ensemble. The most common algorithm to use for speed and model performance is a decision tree with a limited tree depth, such as between 4 and 8 levels.

Q. How many ensemble members should be used?

The number of trees in the ensemble should be tuned based on the specifics of the dataset and other hyperparameters such as the learning rate.

Q. Won't the ensemble overfit with too many trees?

Yes, gradient boosting models can overfit. It is important to carefully choose model hyperparameters using a search procedure, such as a grid search. The learning rate, also called shrinkage, can be set to smaller values in order to slow down the rate of learning with the increase of the number of models used in the ensemble and in turn reduce the effect of overfitting.

Q. What are the downsides of gradient boosting?

Gradient boosting can be challenging to configure, often requiring a grid search or similar search procedure. It can be very slow to train a gradient boosting model as trees must be added sequentially, unlike bagging and stacking based models where ensemble members can be trained in parallel.

Q. What problems are well suited to gradient boosting?

Gradient boosting performs well on a wide range of regression and classification predictive modeling problems. It might be one of the most popular algorithms for structured data (tabular data) given that it performs so well on average.

22.7 Further Reading

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

Papers

- *Arcing the Edge*, 1998.
  http://statistics.berkeley.edu/sites/default/files/tech-reports/486.pdf
- *Stochastic Gradient Boosting*, 1999.
  https://statweb.stanford.edu/~jhf/ftp/stobst.pdf
- *Boosting Algorithms as Gradient Descent in Function Space*, 1999.
  http://papers.nips.cc/paper/1766-boosting-algorithms-as-gradient-descent.pdf

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

APIs

- `sklearn.ensemble.GradientBoostingRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html
- `sklearn.ensemble.GradientBoostingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

Articles

- Gradient boosting, Wikipedia.
  https://en.wikipedia.org/wiki/Gradient_boosting

22.8 Summary

In this tutorial, you discovered how to develop Gradient Boosting ensembles for classification and regression. Specifically, you learned:

- Gradient Boosting ensemble is an ensemble created from decision trees added sequentially to the model.
- How to use the Gradient Boosting ensemble for classification and regression with scikit-learn.
- How to explore the effect of Gradient Boosting model hyperparameters on model performance.

Next

In the next section, we will take a closer look at an extension to gradient boosting called the extreme gradient boosting ensemble.

Chapter 23

Extreme Gradient Boosting Ensemble

Extreme Gradient Boosting (XGBoost) is an open-source library that provides an efficient and effective implementation of the gradient boosting algorithm. Although other open-source implementations of the approach existed before XGBoost, the release of XGBoost appeared to unleash the power of the technique and made the applied machine learning community take notice of gradient boosting more generally. Shortly after its development and initial release, XGBoost became the go-to method and often the key component in winning solutions for classification and regression problems in machine learning competitions. In this tutorial, you will discover how to develop Extreme Gradient Boosting ensembles for classification and regression. After completing this tutorial, you will know:

- Extreme Gradient Boosting is an efficient open-source implementation of the stochastic gradient boosting ensemble algorithm.
- How to develop XGBoost ensembles for classification and regression with the scikit-learn API.
- How to explore the effect of XGBoost model hyperparameters on model performance.

Let's get started.

23.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Extreme Gradient Boosting Algorithm
2. Evaluate XGBoost Ensembles
3. XGBoost Hyperparameters

23.2 Extreme Gradient Boosting Algorithm

Gradient boosting refers to a class of ensemble machine learning algorithms that can be used for classification or regression predictive modeling problems (introduced in Chapter 22). Ensembles

are constructed from decision tree models. Trees are added one at a time to the ensemble and fit to correct the prediction errors made by prior models. This is a type of ensemble machine learning model referred to as boosting. Models are fit using any arbitrary differentiable loss function and gradient descent optimization algorithm. This gives the technique its name, *gradient boosting*, as the loss gradient is minimized as the model is fit, much like a neural network.

Extreme Gradient Boosting, or XGBoost for short is an efficient open-source implementation of the gradient boosting algorithm. As such, XGBoost is an algorithm, an open-source project, and a Python library. It was initially developed by Tianqi Chen and was described by Chen and Carlos Guestrin in their 2016 paper titled *XGBoost: A Scalable Tree Boosting System*. It is designed to be both computationally efficient (e.g. fast to execute) and highly effective, perhaps more effective than other open-source implementations. The two main reasons to use XGBoost are execution speed and model performance. Generally, XGBoost is fast when compared to other implementations of gradient boosting. Szilard Pafka performed some objective benchmarks comparing the performance of XGBoost to other implementations of gradient boosting and bagged decision trees. He wrote up his results in May 2015 in the blog post titled *Benchmarking Random Forest Implementations*. His results showed that XGBoost was almost always faster than the other benchmarked implementations from R, Python Spark, and H2O. From his experiment, he commented:

> I also tried XGBoost, a popular library for boosting which is capable of building random forests as well. It is fast, memory efficient and of high accuracy
>
> — *Benchmarking Random Forest Implementations, Szilard Pafka*, 2015.

XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems. The evidence is that it is the go-to algorithm for competition winners on the Kaggle competitive data science platform.

> Among the 29 challenge winning solutions 3 published at Kaggle's blog during 2015, 17 solutions used XGBoost. [...] The success of the system was also witnessed in KDDCup 2015, where XGBoost was used by every winning team in the top-10.
>
> — *XGBoost: A Scalable Tree Boosting System*, 2016.

Now that we are familiar with what XGBoost is and why it is important, let's take a closer look at how we can use it in our predictive modeling projects.

23.3 Evaluate XGBoost Ensembles

XGBoost can be installed as a standalone library and an XGBoost model can be developed using the scikit-learn API. The first step is to install the XGBoost library if it is not already installed. This can be achieved using the `pip` Python package manager on most platforms; for example:

```
pip install xgboost
```

Listing 23.1: Example installing the XGBoost library.

You can then confirm that the XGBoost library was installed correctly and can be used by running the following script.

```
# check xgboost library version
import xgboost
print(xgboost.__version__)
```

Listing 23.2: Example of checking the XGBoost library version.

Running the script will print your version of the XGBoost library you have installed. Your version should be the same or higher. If not, you must upgrade your version of the XGBoost library.

```
1.1.1
```

Listing 23.3: Example output from checking the XGBoost library version.

It is possible that you may have problems with the latest version of the library. It is not your fault. Sometimes, the most recent version of the library imposes additional requirements or may be less stable. If you do have errors when trying to run the above script, I recommend downgrading to version 1.0.2 (or lower). This can be achieved by specifying the version to install to the `pip` command, as follows:

```
pip install xgboost==1.0.2
```

Listing 23.4: Example installing a specific version of the XGBoost library.

If you see a warning message, you can safely ignore it for now. For example, below is an example of a warning message that you may see and can ignore:

```
FutureWarning: pandas.util.testing is deprecated. Use the functions in the public API at
    pandas.testing instead.
```

Listing 23.5: Example a warning that can be safely ignored.

The XGBoost library has its own custom API, although we will use the method via the scikit-learn wrapper classes: `XGBRegressor` and `XGBClassifier`. This will allow us to use the full suite of tools from the scikit-learn machine learning library to prepare data and evaluate models. Both models operate the same way and take the same arguments that influence how the decision trees are created and added to the ensemble. Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model.

When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions. Let's take a look at how to develop an XGBoost ensemble for both classification and regression.

23.3.1 XGBoost Ensemble for Classification

In this section, we will look at using XGBoost for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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 23.6: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 23.7: Example output from creating the synthetic classification dataset.

Next, we can evaluate an XGBoost model on this dataset. We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate xgboost algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from xgboost import XGBClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = XGBClassifier()
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 23.8: Example of evaluating an XGBoost ensemble the synthetic classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the XGBoost ensemble with default hyperparameters achieves a classification accuracy of about 92.5 percent on this test dataset.

```
Mean Accuracy: 0.925 (0.028)
```

Listing 23.9: Example output from evaluating an XGBoost ensemble the synthetic classification dataset.

We can also use the XGBoost model as a final model and make predictions for classification. First, the XGBoost ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. Depending on the library version, this function may expect data to be provided as a NumPy array. The example below demonstrates this on our binary classification dataset.

```
# make predictions using xgboost for classification
from numpy import asarray
from sklearn.datasets import make_classification
from xgboost import XGBClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = XGBClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719, 0.28422388,
    -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799, 3.34692332,
    4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict(asarray([row]))
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 23.10: Example of making a prediction with an XGBoost ensemble for classification.

Running the example fits the XGBoost ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 23.11: Example output from making a prediction with an XGBoost ensemble for classification.

Now that we are familiar with using XGBoost for classification, let's look at the API for regression.

23.3.2 XGBoost Ensemble for Regression

In this section, we will look at using XGBoost for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 23.12: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 23.13: Example output from creating the synthetic regression dataset.

Next, we can evaluate an XGBoost algorithm on this dataset. As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate xgboost ensemble for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from xgboost import XGBRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# define the model
model = XGBRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 23.14: Example of evaluating an XGBoost ensemble the synthetic regression dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the XGBoost ensemble with default hyperparameters achieves a MAE of about 76.

```
MAE: -76.447 (3.859)
```

Listing 23.15: Example output from evaluating an XGBoost ensemble the synthetic regression dataset.

We can also use the XGBoost model as a final model and make predictions for regression. First, the XGBoost ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. As with classification, the single row of data must be represented as a two-dimensional matrix in NumPy array format. The example below demonstrates this on our regression dataset.

```
# make predictions using xgboost for regression
from numpy import asarray
from sklearn.datasets import make_regression
from xgboost import XGBRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# define the model
model = XGBRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.20543991, -0.97049844, -0.81403429, -0.23842689, -0.60704084, -0.48541492,
    0.53113006, 2.01834338, -0.90745243, -1.85859731, -1.02334791, -0.6877744, 0.60984819,
    -0.70630121, -1.29161497, 1.32385441, 1.42150747, 1.26567231, 2.56569098, -0.11154792]
yhat = model.predict(asarray([row]))
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Listing 23.16: Example of making a prediction with an XGBoost ensemble for regression.

Running the example fits the XGBoost ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: 50
```

Listing 23.17: Example output from making a prediction with an XGBoost ensemble for regression.

Now that we are familiar with using the XGBoost Scikit-Learn API to evaluate and use XGBoost ensembles, let's look at configuring the model.

23.4 XGBoost Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the XGBoost ensemble and their effect on model performance.

23.4.1 Explore Number of Trees

An important hyperparameter for the XGBoost ensemble algorithm is the number of decision trees used in the ensemble. Recall that decision trees are added to the model sequentially in an effort to correct and improve upon the predictions made by prior trees. As such, more trees are often better. The number of trees can be set via the `n_estimators` argument and defaults to 100. The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore xgboost number of trees effect on performance
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 xgboost import XGBClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define the number of trees to explore
  trees = [10, 50, 100, 500, 1000, 5000]
  for n in trees:
    models[str(n)] = XGBClassifier(n_estimators=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 23.18: Example of evaluating the effect of varying the number of trees on the XGBoost ensemble.

Running the example first reports the mean accuracy for each configured number of decision trees.

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

In this case, we can see that performance improves on this dataset until about 500 trees, after which performance appears to level off or decrease.

```
>10 0.885 (0.029)
>50 0.915 (0.029)
>100 0.925 (0.028)
>500 0.927 (0.028)
>1000 0.926 (0.028)
>5000 0.925 (0.027)
```

Listing 23.19: Example output from evaluating the effect of varying the number of trees on the XGBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees. We can see the general trend of increasing model performance and ensemble size.

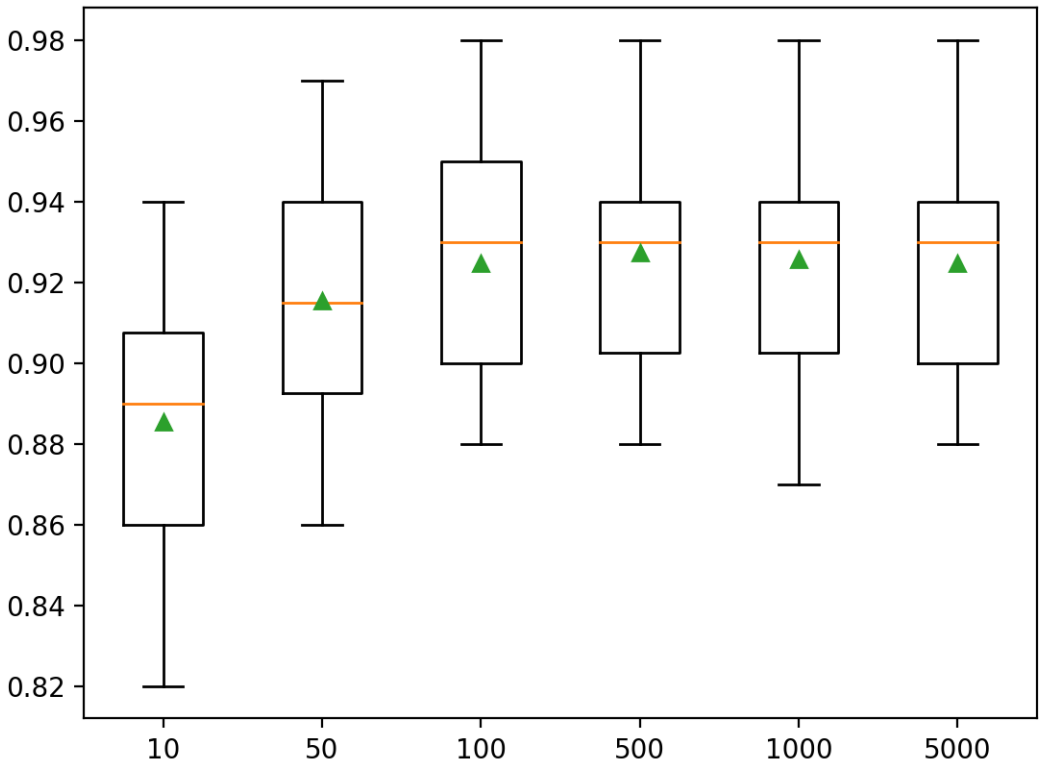

Figure 23.1: Box Plots of XGBoost Ensemble Size vs. Classification Accuracy.

23.4.2 Explore Tree Depth

Varying the depth of each tree added to the ensemble is another important hyperparameter for gradient boosting. The tree depth controls how specialized each tree is to the training dataset: how general or overfit it might be. Trees are preferred that are not too shallow and general (like AdaBoost) and not too deep and specialized (like bootstrap aggregation). Gradient boosting generally performs well with trees that have a modest depth, finding a balance between skill and generality. Tree depth is controlled via the `max_depth` argument and defaults to 6. The example below explores tree depths between 1 and 10 and the effect on model performance.

```
# explore xgboost tree depth effect on performance
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 xgboost import XGBClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore tree depth from 1 to 10
  for i in range(1,11):
    models[str(i)] = XGBClassifier(max_depth=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
```

```
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 23.20: Example of evaluating the effect of varying the tree depth on the XGBoost ensemble.

Running the example first reports the mean accuracy for each configured tree depth.

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

In this case, we can see that performance improves with tree depth, perhaps peeking around a depth of 3 to 8, after which the deeper, more specialized trees result in worse performance.

```
>1 0.849 (0.028)
>2 0.906 (0.032)
>3 0.926 (0.027)
>4 0.930 (0.027)
>5 0.924 (0.031)
>6 0.925 (0.028)
>7 0.926 (0.030)
>8 0.926 (0.029)
>9 0.921 (0.032)
>10 0.923 (0.035)
```

Listing 23.21: Example output from evaluating the effect of varying the tree depth on the XGBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured tree depth. We can see the general trend of increasing model performance with the tree depth to a point, after which performance begins to sit flat or degrade with the over-specialized trees.

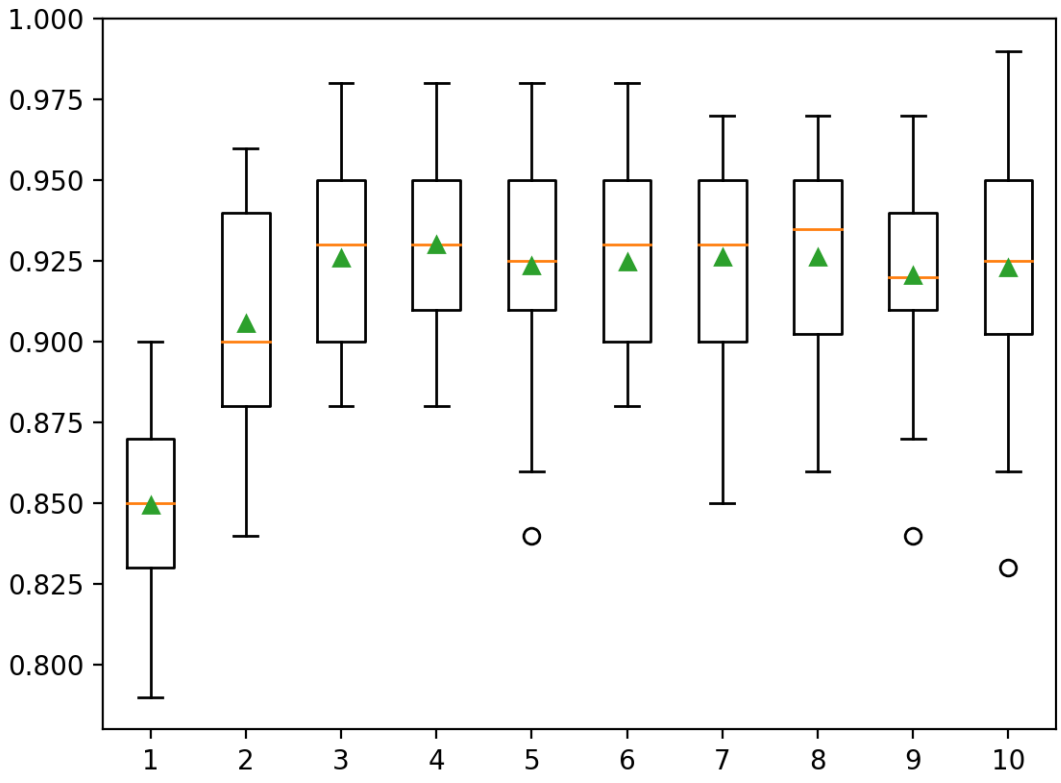

Figure 23.2: Box Plots of XGBoost Ensemble Tree Depth vs. Classification Accuracy.

23.4.3 Explore Learning Rate

Learning rate controls the amount of contribution that each model has on the ensemble prediction. Smaller rates may require more decision trees in the ensemble. The learning rate can be controlled via the `eta` argument and defaults to 0.3. The example below explores the learning rate and compares the effect of values between 0.0001 and 1.0.

```
# explore xgboost learning rate effect on performance
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 xgboost import XGBClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
```

```
def get_models():
  models = dict()
  # define learning rates to explore
  rates = [0.0001, 0.001, 0.01, 0.1, 1.0]
  for r in rates:
    key = '%.4f' % r
    models[key] = XGBClassifier(eta=r)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 23.22: Example of evaluating the effect of varying the learning rate on the XGBoost ensemble.

Running the example first reports the mean accuracy for each configured learning rate.

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

In this case, we can see that a larger learning rate results in better performance on this dataset. We would expect that adding more trees to the ensemble for the smaller learning rates would further lift performance. This highlights the trade-off between the number of trees (speed of training) and learning rate, e.g. we can fit a model faster by using fewer trees and a larger learning rate.

```
>0.0001 0.804 (0.039)
>0.0010 0.814 (0.037)
>0.0100 0.867 (0.027)
>0.1000 0.923 (0.030)
>1.0000 0.913 (0.030)
```

Listing 23.23: Example output from evaluating the effect of varying the learning rate on the XGBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured learning rate. We can see the general trend of increasing model performance with the increase in learning rate of 0.1, after which performance degrades.

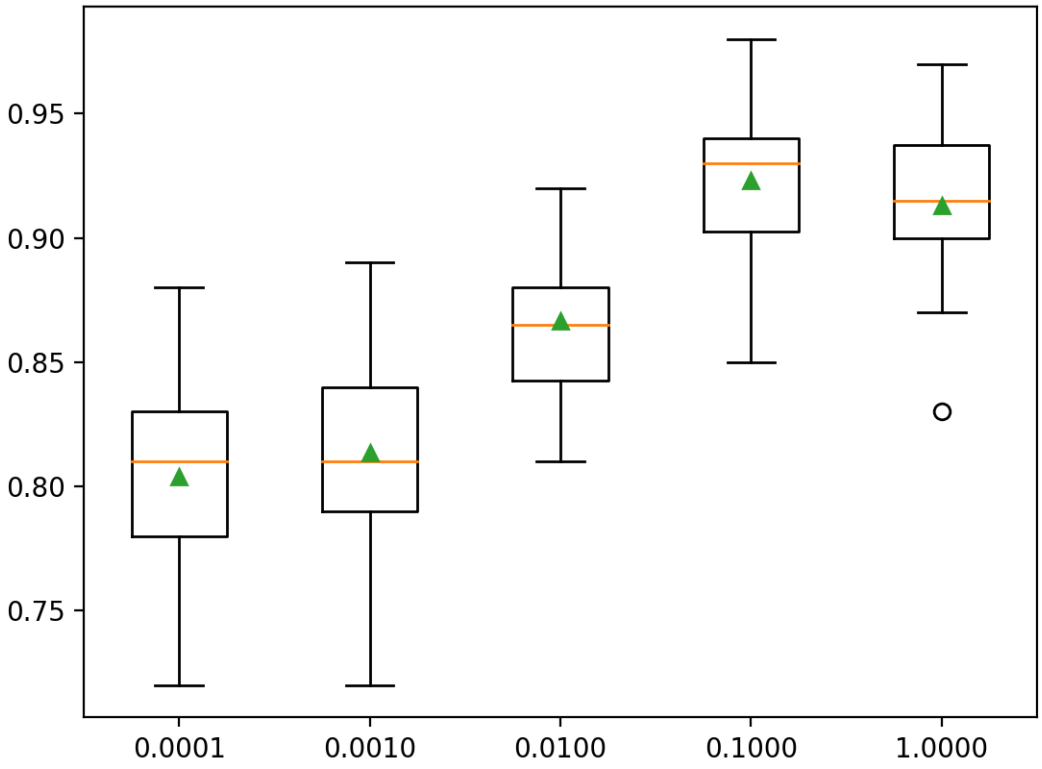

Figure 23.3: Box Plot of XGBoost Learning Rate vs. Classification Accuracy.

23.4.4 Explore Sample Size

The number of instances used to fit each tree can be varied. This means that each tree is fit on a randomly selected subset of the training dataset. Using fewer instances introduces more variance for each tree, although it can improve the overall performance of the model. The sample size used to fit each tree is specified by the `subsample` argument and can be set to a fraction of the training dataset size. By default, it is set to 1.0 to use the entire training dataset. The example below demonstrates the effect of the sample size on model performance with ratios varying from 10 percent to 100 percent in 10 percent increments.

```
# explore xgboost subsample ratio effect on performance
from numpy import arange
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 xgboost import XGBClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore ratio of samples from 10% to 100% in 10% increments
  for i in arange(0.1, 1.1, 0.1):
    key = '%.1f' % i
    models[key] = XGBClassifier(subsample=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 23.24: Example of evaluating the effect of varying the sample size on the XGBoost ensemble.

Running the example first reports the mean accuracy for each configured sample size.

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

In this case, we can see that mean performance is probably best for a sample size that covers most of the dataset, such as 80 percent or higher.

```
>0.1 0.876 (0.027)
>0.2 0.912 (0.033)
>0.3 0.917 (0.032)
>0.4 0.925 (0.026)
>0.5 0.928 (0.027)
>0.6 0.926 (0.024)
>0.7 0.925 (0.031)
>0.8 0.928 (0.028)
>0.9 0.928 (0.025)
>1.0 0.925 (0.028)
```

Listing 23.25: Example output from evaluating the effect of varying the sample size on the XGBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured sample size ratio. We can see the general trend of increasing model performance, perhaps peaking around 80 percent and staying somewhat level.

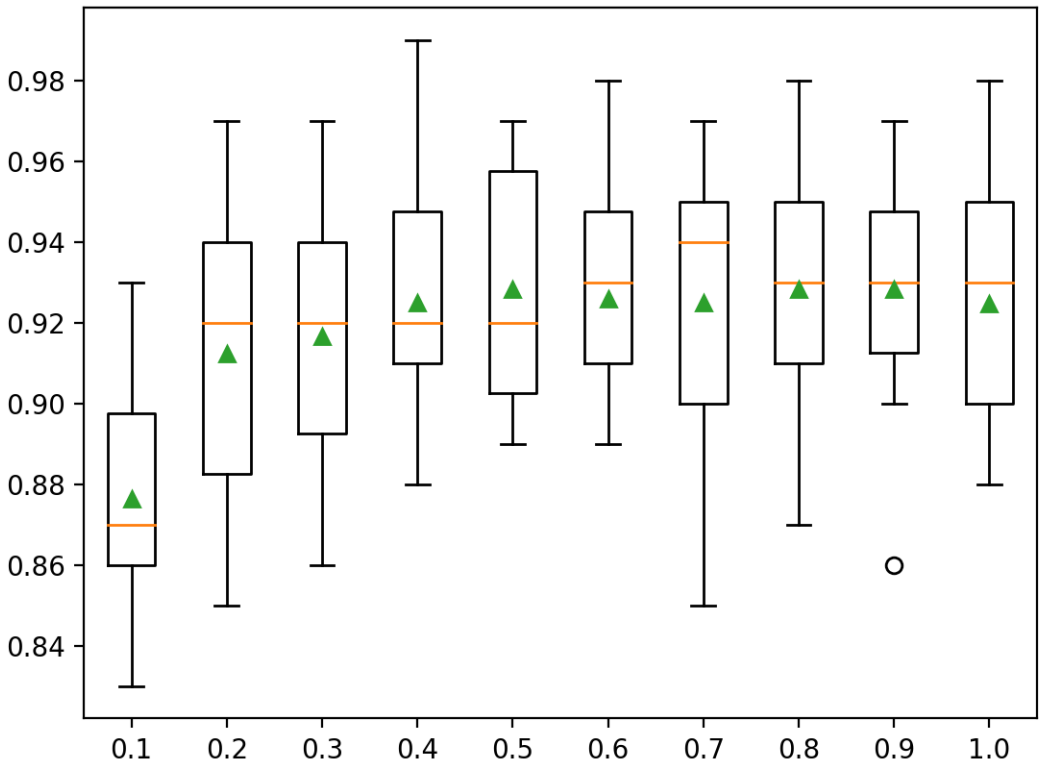

Figure 23.4: Box Plots of XGBoost Ensemble Sample Ratio vs. Classification Accuracy.

23.4.5 Explore Number of Features

The number of features used to fit each decision tree can be varied. Like changing the sample size, changing the number of features introduces additional variance into the model, which may improve performance, although it might require an increase in the number of trees. The number of features used by each tree is taken as a random sample and is specified by the `colsample_bytree` argument and defaults to all features in the training dataset, e.g. 100 percent or a value of 1.0. You can also sample columns for each split, and this is controlled by the `colsample_bylevel` argument, but we will not look at this hyperparameter in this tutorial. The example below explores the effect of the number of features on model performance with ratios varying from 10 percent to 100 percent in 10 percent increments.

```
# explore xgboost column ratio per tree effect on performance
from numpy import arange
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 xgboost import XGBClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore ratio of features from 10% to 100% in 10% increments
  for i in arange(0.1, 1.1, 0.1):
    key = '%.1f' % i
    models[key] = XGBClassifier(colsample_bytree=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
```

```
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 23.26: Example of evaluating the effect of varying the number of features on the XGBoost ensemble.

Running the example first reports the mean accuracy for each configured ratio of columns.

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

In this case, we can see that mean performance increases to about half the number of features (50 percent) and stays somewhat level after that. It's surprising that removing half of the input variables per tree has so little effect.

```
>0.1 0.861 (0.033)
>0.2 0.906 (0.027)
>0.3 0.923 (0.029)
>0.4 0.917 (0.029)
>0.5 0.928 (0.030)
>0.6 0.929 (0.031)
>0.7 0.924 (0.027)
>0.8 0.931 (0.025)
>0.9 0.927 (0.033)
>1.0 0.925 (0.028)
```

Listing 23.27: Example output from evaluating the effect of varying the features of samples on the XGBoost ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured column ratio. We can see the general trend of increasing model performance perhaps peaking with a ratio of 60 percent and staying somewhat level.

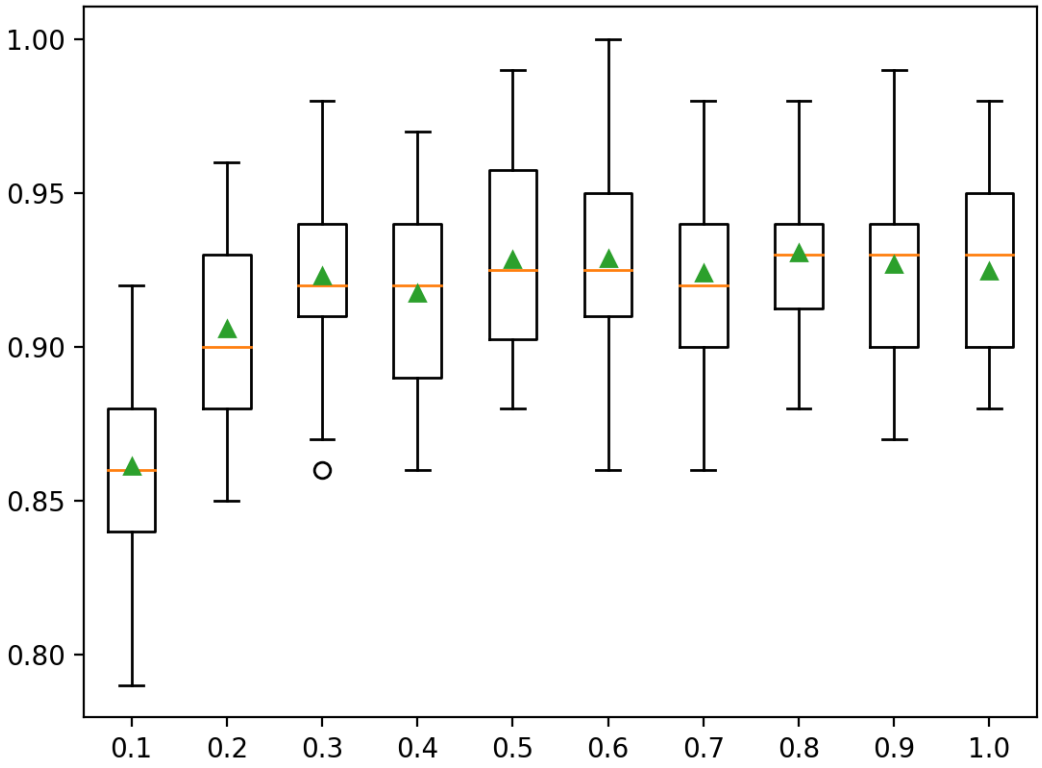

Figure 23.5: Box Plots of XGBoost Ensemble Column Ratio vs. Classification Accuracy.

23.5 Further Reading

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

Papers

- *XGBoost: A Scalable Tree Boosting System*, 2016.
  https://arxiv.org/abs/1603.02754

Project

- XGBoost Project, GitHub.
  https://github.com/dmlc/xgboost
- XGBoost Documentation
  https://xgboost.readthedocs.io/en/latest/

APIs

- XGBoost Installation Guide
  https://xgboost.readthedocs.io/en/latest/build.html
- `xgboost.XGBRegressor` API.
  https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBRegressor
- `xgboost.XGBClassifier` API.
  https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier

Articles

- Gradient boosting, Wikipedia.
  https://en.wikipedia.org/wiki/Gradient_boosting
- XGBoost, Wikipedia.
  https://en.wikipedia.org/wiki/XGBoost
- *Benchmarking Random Forest Implementations, Szilard Pafka*, 2015.
  http://datascience.la/benchmarking-random-forest-implementations/

23.6 Summary

In this tutorial, you discovered how to develop Extreme Gradient Boosting ensembles for classification and regression. Specifically, you learned:

- Extreme Gradient Boosting is an efficient open-source implementation of the stochastic gradient boosting ensemble algorithm.
- How to develop XGBoost ensembles for classification and regression with the scikit-learn API.
- How to explore the effect of XGBoost model hyperparameters on model performance.

Next

In the next section, we will take a closer look at an extension to gradient boosting called the light gradient boosting machine ensemble.

Chapter 24

Light Gradient Boosting Machine Ensemble

Light Gradient Boosted Machine, or LightGBM for short, is an open-source library that provides an efficient and effective implementation of the gradient boosting algorithm. LightGBM extends the gradient boosting algorithm by adding a type of automatic feature selection as well as focusing on boosting examples with larger gradients. This can result in a dramatic speedup of training and improved predictive performance.

As such, LightGBM has become a de facto algorithm for machine learning competitions when working with tabular data for regression and classification predictive modeling tasks. As such, it owns a share of the blame for the increased popularity and wider adoption of gradient boosting methods in general, along with Extreme Gradient Boosting (XGBoost). In this tutorial, you will discover how to develop Light Gradient Boosted Machine ensembles for classification and regression. After completing this tutorial, you will know:

- Light Gradient Boosted Machine (LightGBM) is an efficient open-source implementation of the stochastic gradient boosting ensemble algorithm.
- How to develop LightGBM ensembles for classification and regression with the scikit-learn API.
- How to explore the effect of LightGBM model hyperparameters on model performance.

Let's get started.

24.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Light Gradient Boosted Machine Algorithm
2. Evaluate LightGBM Ensembles
3. LightGBM Hyperparameters

24.2 Light Gradient Boosted Machine Algorithm

Gradient boosting refers to a class of ensemble machine learning algorithms that can be used for classification or regression predictive modeling problems (introduced in Chapter 22). Ensembles are constructed from decision tree models. Trees are added one at a time to the ensemble and fit to correct the prediction errors made by prior models. This is a type of ensemble machine learning model referred to as boosting. Models are fit using any arbitrary differentiable loss function and gradient descent optimization algorithm. This gives the technique its name, *gradient boosting*, as the loss gradient is minimized as the model is fit, much like a neural network.

Light Gradient Boosted Machine, or LightGBM for short, is an open-source implementation of gradient boosting designed to be efficient and perhaps more effective than other implementations. As such, LightGBM refers to the open-source project, the software library, and the machine learning algorithm. In this way, it is very similar to the Extreme Gradient Boosting or XGBoost technique. LightGBM was described by Guolin Ke, et al. in the 2017 paper titled *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*. The implementation introduces two key ideas: GOSS and EFB.

Gradient-based One-Side Sampling, or GOSS for short, is a modification to the gradient boosting method that focuses attention on those training examples that result in a larger gradient, in turn speeding up learning and reducing the computational complexity of the method.

> With GOSS, we exclude a significant proportion of data instances with small gradients, and only use the rest to estimate the information gain. We prove that, since the data instances with larger gradients play a more important role in the computation of information gain, GOSS can obtain quite accurate estimation of the information gain with a much smaller data size.
>
> — *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, 2017.

Exclusive Feature Bundling, or EFB for short, is an approach for bundling sparse (mostly zero) mutually exclusive features, such as categorical variable inputs that have been one hot encoded. As such, it is a type of automatic feature selection.

> ... we bundle mutually exclusive features (i.e., they rarely take nonzero values simultaneously), to reduce the number of features.
>
> — *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, 2017.

Together, these two changes can accelerate the training time of the algorithm by up to 20x. As such, LightGBM may be considered gradient boosting decision trees (GBDT) with the addition of GOSS and EFB.

> We call our new GBDT implementation with GOSS and EFB LightGBM. Our experiments on multiple public datasets show that, LightGBM speeds up the training process of conventional GBDT by up to over 20 times while achieving almost the same accuracy
>
> — *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, 2017.

24.3 Evaluate LightGBM Ensembles

LightGBM can be installed as a standalone library and the LightGBM model can be developed using the scikit-learn API. The first step is to install the LightGBM library, if it is not already installed. This can be achieved using the `pip` Python package manager on most platforms; for example:

```
pip install lightgbm
```

Listing 24.1: Example installing the LightGBM library.

You can then confirm that the LightGBM library was installed correctly and can be used by running the following script.

```
# check lightgbm library version
import lightgbm
print(lightgbm.__version__)
```

Listing 24.2: Example of checking the LightGBM library version.

Running the script will print your version of the LightGBM library you have installed. Your version should be the same or higher. If not, you must upgrade your version of the LightGBM library.

```
2.3.1
```

Listing 24.3: Example output from checking the LightGBM library version.

The LightGBM library has its own custom API, although we will use the method via the scikit-learn wrapper classes: `LGBMRegressor` and `LGBMClassifier`. This will allow us to use the full suite of tools from the scikit-learn machine learning library to prepare data and evaluate models. Both models operate the same way and take the same arguments that influence how the decision trees are created and added to the ensemble.

Randomness is used in the construction of the model. This means that each time the algorithm is run on the same data, it will produce a slightly different model. When using machine learning algorithms that have a stochastic learning algorithm, it is good practice to evaluate them by averaging their performance across multiple runs or repeats of cross-validation. When fitting a final model, it may be desirable to either increase the number of trees until the variance of the model is reduced across repeated evaluations, or to fit multiple final models and average their predictions. Let's take a look at how to develop a LightGBM ensemble for both classification and regression.

24.3.1 LightGBM Ensemble for Classification

In this section, we will look at using LightGBM for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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 24.4: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 24.5: Example output from creating the synthetic classification dataset.

Next, we can evaluate a LightGBM algorithm on this dataset. We will evaluate the model using repeated stratified k-fold cross-validation with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

```
# evaluate lightgbm algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from lightgbm import LGBMClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = LGBMClassifier()
# define the evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model on the dataset
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# report performance
print('Mean Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 24.6: Example of evaluating an LightGBM ensemble the synthetic classification dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the LightGBM ensemble with default hyperparameters achieves a classification accuracy of about 92.5 percent on this test dataset.

```
Mean Accuracy: 0.925 (0.031)
```

Listing 24.7: Example output from evaluating an LightGBM ensemble the synthetic classification dataset.

We can also use the LightGBM model as a final model and make predictions for classification. First, the LightGBM ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make predictions using lightgbm for classification
from sklearn.datasets import make_classification
from lightgbm import LGBMClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# define the model
model = LGBMClassifier()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.2929949, -4.21223056, -1.288332, -2.17849815, -0.64527665, 2.58097719, 0.28422388,
    -7.1827928, -1.91211104, 2.73729512, 0.81395695, 3.96973717, -2.66939799, 3.34692332,
    4.19791821, 0.99990998, -0.30201875, -4.43170633, -2.82646737, 0.44916808]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % yhat[0])
```

Listing 24.8: Example of making a prediction with an LightGBM ensemble for classification.

Running the example fits the LightGBM ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 24.9: Example output from making a prediction with an LightGBM ensemble for classification.

Now that we are familiar with using LightGBM for classification, let's look at the API for regression.

24.3.2 LightGBM Ensemble for Regression

In this section, we will look at using LightGBM for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 24.10: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 24.11: Example output from creating the synthetic regression dataset.

Next, we can evaluate a LightGBM algorithm on this dataset. As we did with the last section, we will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# evaluate lightgbm ensemble for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from lightgbm import LGBMRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# define the model
model = LGBMRegressor()
# define the evaluation procedure
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the model
n_scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
# report performance
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Listing 24.12: Example of evaluating an LightGBM ensemble the synthetic regression dataset.

Running the example reports the mean and standard deviation accuracy of the model.

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

In this case, we can see the LightGBM ensemble with default hyperparameters achieves a MAE of about 60.

```
MAE: -60.004 (2.887)
```

Listing 24.13: Example output from evaluating an LightGBM ensemble the synthetic regression dataset.

We can also use the LightGBM model as a final model and make predictions for regression. First, the LightGBM ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# make predictions using lightgbm for regression
from sklearn.datasets import make_regression
from lightgbm import LGBMRegressor
# define dataset
```

```
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=7)
# define the model
model = LGBMRegressor()
# fit the model on the whole dataset
model.fit(X, y)
# make a single prediction
row = [0.20543991, -0.97049844, -0.81403429, -0.23842689, -0.60704084, -0.48541492,
    0.53113006, 2.01834338, -0.90745243, -1.85859731, -1.02334791, -0.6877744, 0.60984819,
    -0.70630121, -1.29161497, 1.32385441, 1.42150747, 1.26567231, 2.56569098, -0.11154792]
yhat = model.predict([row])
# summarize prediction
print('Prediction: %d' % yhat[0])
```

Listing 24.14: Example of making a prediction with an LightGBM ensemble for regression.

Running the example fits the LightGBM ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Prediction: 52
```

Listing 24.15: Example output from making a prediction with an LightGBM ensemble for regression.

Now that we are familiar with using the scikit-learn API to evaluate and use LightGBM ensembles, let's look at configuring the model.

24.4 LightGBM Hyperparameters

In this section, we will take a closer look at some of the hyperparameters you should consider tuning for the LightGBM ensemble and their effect on model performance. There are many hyperparameters we can look at for LightGBM, although in this case, we will look at the number of trees and tree depth, the learning rate, and the boosting type.

24.4.1 Explore Number of Trees

An important hyperparameter for the LightGBM ensemble algorithm is the number of decision trees used in the ensemble. Recall that decision trees are added to the model sequentially in an effort to correct and improve upon the predictions made by prior trees. As such, more trees are often better. The number of trees can be set via the `n_estimators` argument and defaults to 100. The example below explores the effect of the number of trees with values between 10 to 5,000.

```
# explore lightgbm number of trees effect on performance
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 lightgbm import LGBMClassifier
from matplotlib import pyplot
```

```
# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define the number of trees to explore
  trees = [10, 50, 100, 500, 1000, 5000]
  for n in trees:
    models[str(n)] = LGBMClassifier(n_estimators=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 24.16: Example of evaluating the effect of varying the number of trees on the LightGBM ensemble.

Running the example first reports the mean accuracy for each configured number of decision trees.

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

In this case, we can see that performance improves on this dataset until about 500 trees, after which performance appears to level off.

```
>10 0.857 (0.033)
```

```
>50 0.916 (0.032)
>100 0.925 (0.031)
>500 0.938 (0.026)
>1000 0.938 (0.028)
>5000 0.937 (0.028)
```

Listing 24.17: Example output from evaluating the effect of varying the number of trees on the LightGBM ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured number of trees. We can see the general trend of increasing model performance and ensemble size.

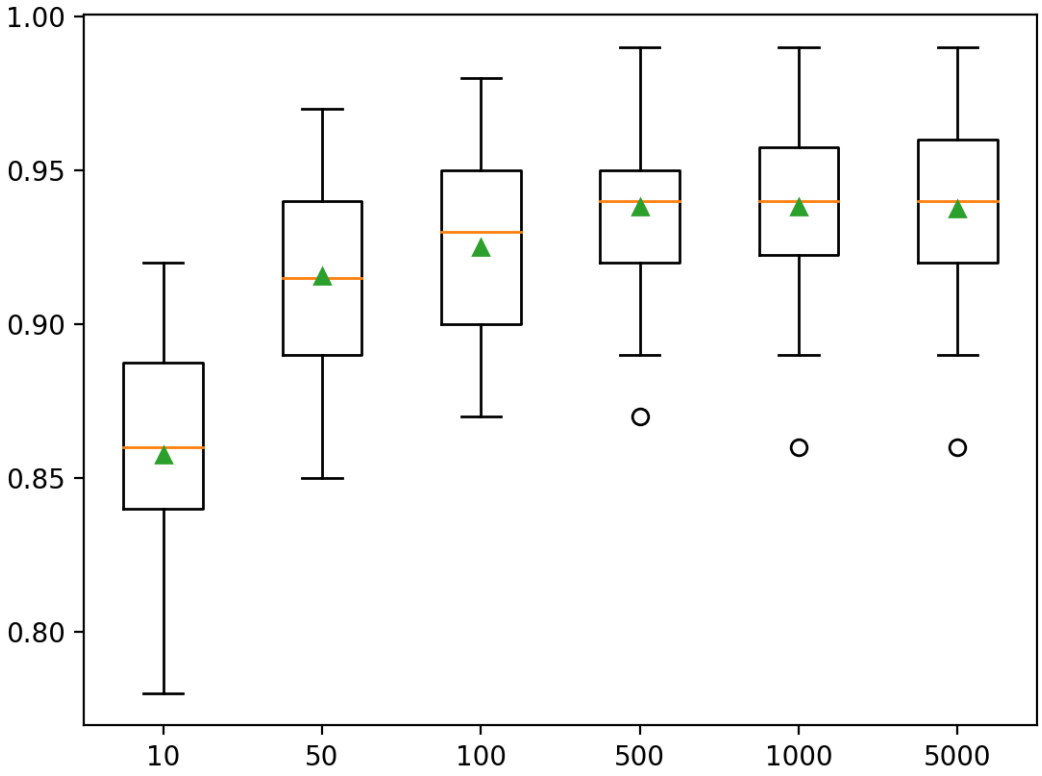

Figure 24.1: Box Plots of LightGBM Ensemble Size vs. Classification Accuracy.

24.4.2 Explore Tree Depth

Varying the depth of each tree added to the ensemble is another important hyperparameter for gradient boosting. The tree depth controls how specialized each tree is to the training dataset: how general or overfit it might be. Trees are preferred that are not too shallow and general (like AdaBoost) and not too deep and specialized (like bootstrap aggregation). Gradient boosting generally performs well with trees that have a modest depth, finding a balance between skill and generality. Tree depth is controlled via the `max_depth` argument and defaults to an unspecified

value as the default mechanism for controlling how complex trees are is to use the number of leaf nodes.

There are two main ways to control tree complexity: the max depth of the trees and the maximum number of terminal nodes (leaves) in the tree. In this case, we are exploring the number of leaves so we need to increase the number of leaves to support deeper trees by setting the `num_leaves` argument. The example below explores tree depths between 1 and 10 and the effect on model performance.

```
# explore lightgbm tree depth effect on performance
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 lightgbm import LGBMClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore tree depths from 1 to 10
  for i in range(1,11):
    models[str(i)] = LGBMClassifier(max_depth=i, num_leaves=2**i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
```

```
pyplot.show()
```

Listing 24.18: Example of evaluating the effect of varying the tree depth on the LightGBM ensemble.

Running the example first reports the mean accuracy for each configured tree depth.

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

In this case, we can see that performance improves with tree depth, perhaps all the way to 10 levels. It might be interesting to explore even deeper trees.

```
>1 0.833 (0.028)
>2 0.870 (0.033)
>3 0.899 (0.032)
>4 0.912 (0.026)
>5 0.925 (0.031)
>6 0.924 (0.029)
>7 0.922 (0.027)
>8 0.926 (0.027)
>9 0.925 (0.028)
>10 0.928 (0.029)
```

Listing 24.19: Example output from evaluating the effect of varying the tree depth on the LightGBM ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured tree depth. We can see the general trend of increasing model performance with the tree depth to a depth of five levels, after which performance begins to sit reasonably flat.

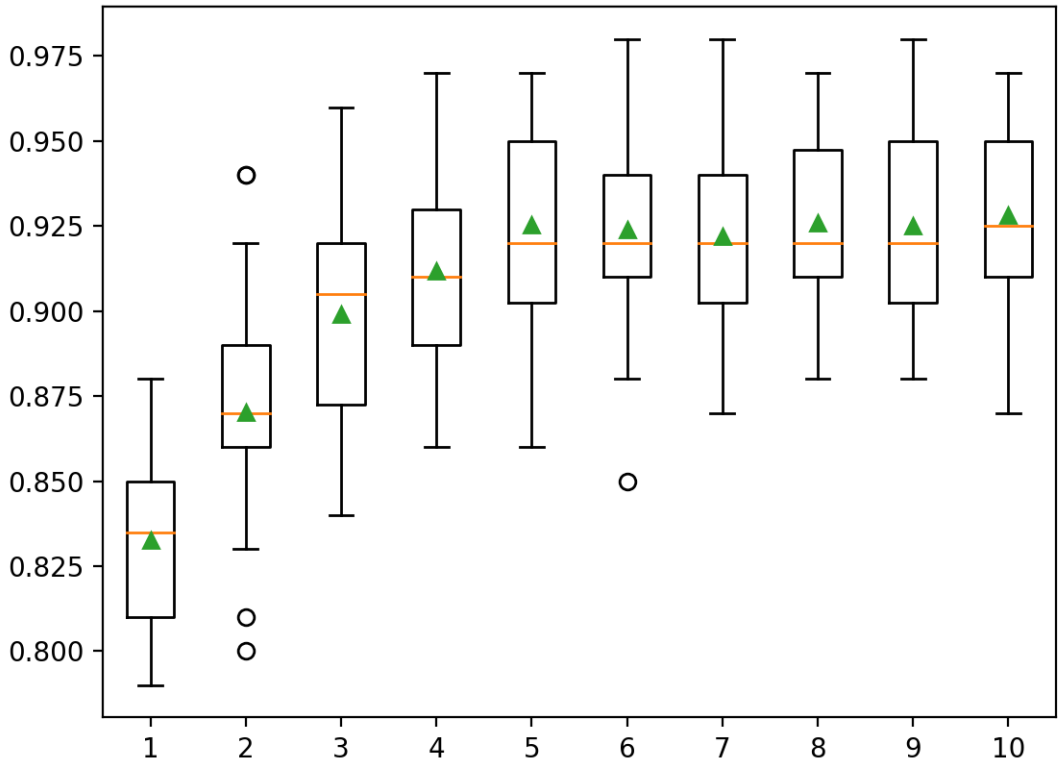

Figure 24.2: Box Plots of LightGBM Ensemble Tree Depth vs. Classification Accuracy.

24.4.3 Explore Learning Rate

Learning rate controls the amount of contribution that each model has on the ensemble prediction. Smaller rates may require more decision trees in the ensemble. The learning rate can be controlled via the `learning_rate` argument and defaults to 0.1. The example below explores the learning rate and compares the effect of values between 0.0001 and 1.0.

```
# explore lightgbm learning rate effect on performance
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 lightgbm import LGBMClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
```

```
def get_models():
  models = dict()
  # define the learning rates to explore
  rates = [0.0001, 0.001, 0.01, 0.1, 1.0]
  for r in rates:
    key = '%.4f' % r
    models[key] = LGBMClassifier(learning_rate=r)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 24.20: Example of evaluating the effect of varying the learning rate on the LightGBM ensemble.

Running the example first reports the mean accuracy for each configured learning rate.

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

In this case, we can see that a larger learning rate results in better performance on this dataset. We would expect that adding more trees to the ensemble for the smaller learning rates would further lift performance.

```
>0.0001 0.800 (0.038)
>0.0010 0.811 (0.035)
>0.0100 0.859 (0.035)
>0.1000 0.925 (0.031)
>1.0000 0.928 (0.025)
```

Listing 24.21: Example output from evaluating the effect of varying the learning rate on the LightGBM ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured learning rate. We can see the general trend of increasing model performance with the increase in learning rate all the way to the large values of 1.0.

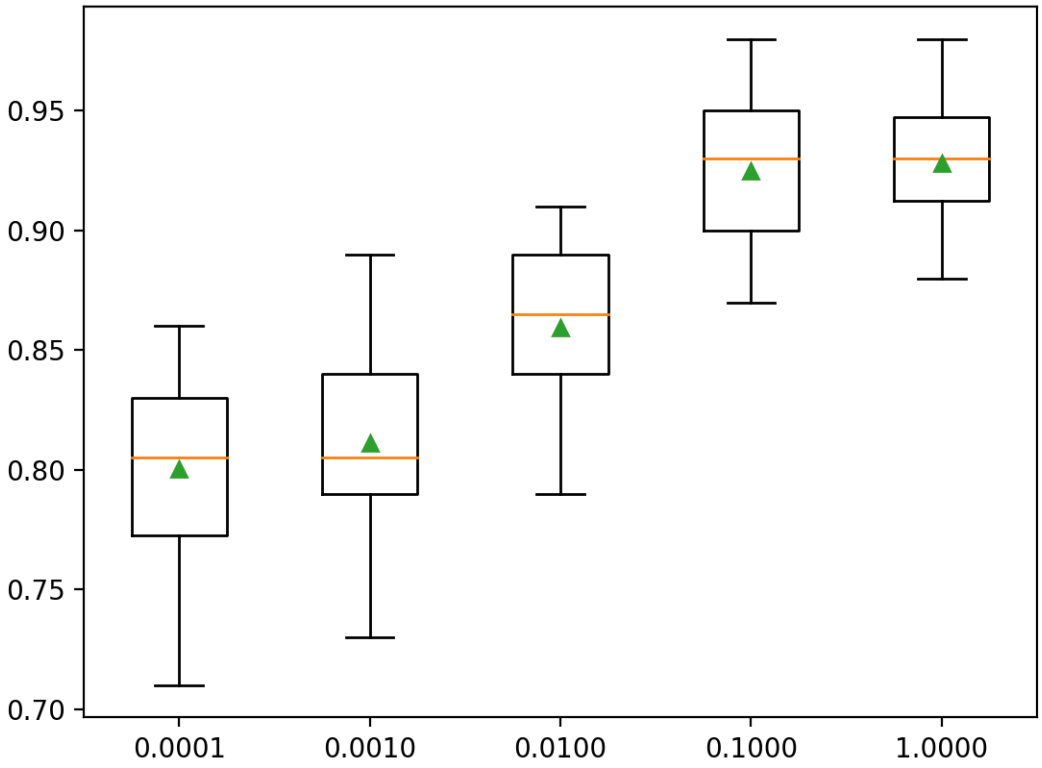

Figure 24.3: Box Plot of LightGBM Learning Rate vs. Classification Accuracy.

24.4.4 Explore Boosting Type

A feature of LightGBM is that it supports a number of different boosting algorithms, referred to as boosting types. The boosting type can be specified via the `boosting_type` argument and take a string to specify the type. The options include:

- `'gbdt'`: Gradient Boosting Decision Tree (GDBT).
- `'dart'`: Dropouts meet Multiple Additive Regression Trees (DART).
- `'goss'`: Gradient-based One-Side Sampling (GOSS).

The default is GDBT, which is the classical gradient boosting algorithm. DART is described in the 2015 paper titled *DART: Dropouts meet Multiple Additive Regression Trees* and, as its name suggests, adds the concept of dropout from deep learning to the Multiple Additive Regression Trees (MART) algorithm, a precursor to gradient boosting decision trees.

> This algorithm is known by many names, including Gradient TreeBoost, boosted trees, and Multiple Additive Regression Trees (MART). We use the latter to refer to this algorithm.
>
> — *DART: Dropouts meet Multiple Additive Regression Trees*, 2015.

GOSS was introduced with the LightGBM paper and library. The approach seeks to only use instances that result in a large error gradient to update the model and drop the rest.

> ... we exclude a significant proportion of data instances with small gradients, and only use the rest to estimate the information gain.
>
> — *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, 2017.

The example below compares LightGBM on the synthetic classification dataset with the three key boosting techniques.

```
# explore lightgbm boosting type effect on performance
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 lightgbm import LGBMClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # define the boosting types to explore
  types = ['gbdt', 'dart', 'goss']
  for t in types:
    models[t] = LGBMClassifier(boosting_type=t)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
```

```
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 24.22: Example of evaluating the effect of varying the boosting type on the LightGBM ensemble.

Running the example first reports the mean accuracy for each configured boosting type.

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

In this case, we can see that the default boosting method performed better than the other two techniques that were evaluated.

```
>gbdt 0.925 (0.031)
>dart 0.912 (0.028)
>goss 0.918 (0.027)
```

Listing 24.23: Example output from evaluating the effect of varying the boosting type on the LightGBM ensemble.

A box and whisker plot is created for the distribution of accuracy scores for each configured boosting method, allowing the techniques to be compared directly.

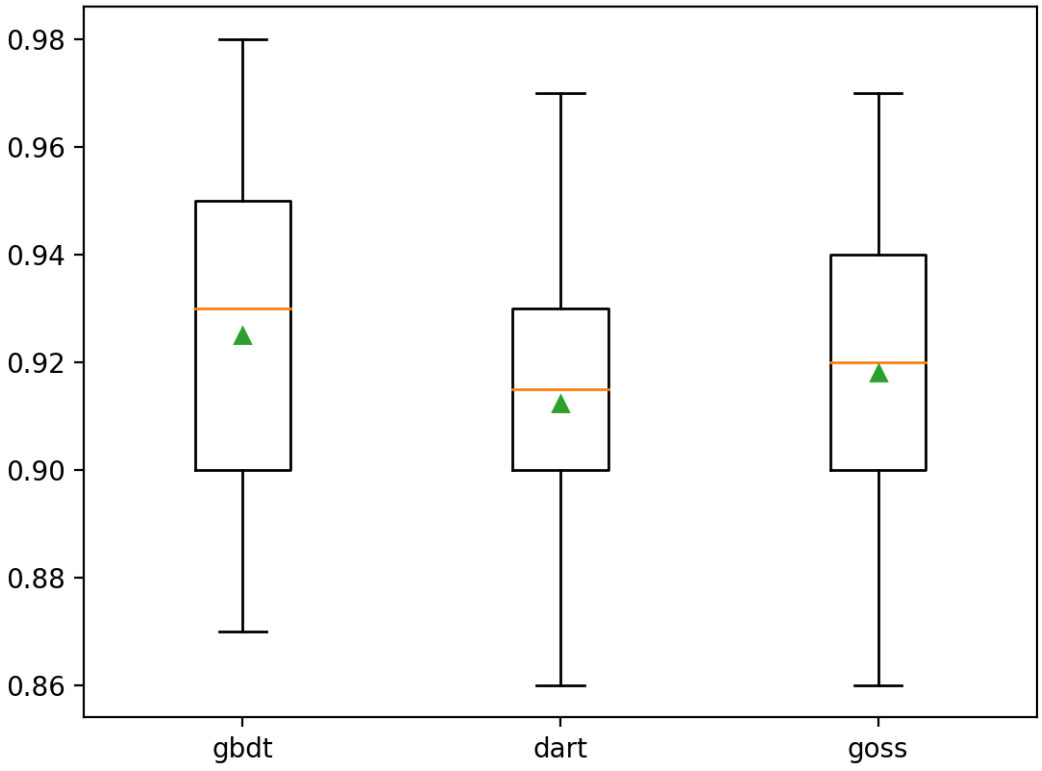

Figure 24.4: Box Plots of LightGBM Boosting Type vs. Classification Accuracy.

24.5 Further Reading

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

Papers

- *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, 2017.
  https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree
- *DART: Dropouts meet Multiple Additive Regression Trees*, 2015.
  https://arxiv.org/abs/1505.01866

APIs

- LightGBM Project, GitHub
  https://github.com/microsoft/LightGBM
- LightGBM's Documentation.
  https://lightgbm.readthedocs.io/

- LightGBM Installation Guide
  https://lightgbm.readthedocs.io/en/latest/Installation-Guide.html
- LightGBM Parameters Tuning.
  https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html
- `lightgbm.LGBMClassifier` API.
  https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html
- `lightgbm.LGBMRegressor` API.
  https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMRegressor.html

Articles

- Gradient boosting, Wikipedia.
  https://en.wikipedia.org/wiki/Gradient_boosting

24.6 Summary

In this tutorial, you discovered how to develop Light Gradient Boosted Machine ensembles for classification and regression. Specifically, you learned:

- Light Gradient Boosted Machine (LightGBM) is an efficient open source implementation of the stochastic gradient boosting ensemble algorithm.
- How to develop LightGBM ensembles for classification and regression with the scikit-learn API.
- How to explore the effect of LightGBM model hyperparameters on model performance.

Next

This was the final chapter in this part, in the next part we will take a closer look at stacking ensembles.

Part VII

Stacking

Chapter 25

Voting Ensemble

Voting is an ensemble machine learning algorithm. For regression, a voting ensemble involves making a prediction that is the average of multiple other regression models. In classification, a hard voting ensemble involves summing the votes for crisp class labels from other models and predicting the class with the most votes. A soft voting ensemble involves summing the predicted probabilities for class labels and predicting the class label with the largest sum probability. In this tutorial, you will discover how to create voting ensembles for machine learning algorithms in Python. After completing this tutorial, you will know:

- A voting ensemble involves summing the predictions made by classification models or averaging the predictions made by regression models.
- How voting ensembles work, when to use voting ensembles, and the limitations of the approach.
- How to implement a hard voting ensemble and soft voting ensemble for classification predictive modeling.

Let's get started.

25.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Voting Ensembles
2. Develop Voting Ensembles
3. Voting Ensemble for Classification
4. Voting Ensemble for Regression

25.2 Voting Ensembles

A voting ensemble (technically called *plurality voting*) is an ensemble machine learning model that combines the predictions from multiple other models (introduced in Chapter 5). It is a technique that may be used to improve model performance, ideally achieving better performance than any single model used in the ensemble. A voting ensemble works by combining the predictions from multiple models. It can be used for classification or regression. In the case of regression, this involves calculating the average of the predictions from the models. In the case of classification, the predictions for each label are taken as votes and summed, where the label with the most votes is predicted.

> ... plurality voting takes the class label which receives the largest number of votes as the final winner.
>
> — Page 73, *Ensemble Methods*, 2012.

There are two approaches to the majority vote prediction for classification; they are hard voting and soft voting. Hard voting involves summing the predictions for each class label and predicting the class label with the most votes. Soft voting involves summing the predicted probabilities (or probability-like scores) for each class label and predicting the class label with the largest probability.

- **Hard Voting**. Predict the class with the largest sum of votes from models.
- **Soft Voting**. Predict the class with the largest summed probability from models.

A voting ensemble may be considered a meta-model, a model of models. As a meta-model, it could be used with any collection of existing trained machine learning models and the existing models do not need to be aware that they are being used in the ensemble. This means you could explore using a voting ensemble on any set or subset of fit models for your predictive modeling task. A voting ensemble is appropriate when you have two or more models that perform well on a predictive modeling task. The models used in the ensemble must mostly agree with their predictions.

> One way to combine outputs is by voting–the same mechanism used in bagging. However, (unweighted) voting only makes sense if the learning schemes perform comparably well. If two of the three classifiers make predictions that are grossly incorrect, we will be in trouble!
>
> — Page 497, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

In fact, if the contributing models were independent, then a majority voting ensemble would be optimal way to combine the models. Unfortunately, contributing models are rarely if ever independent as they are trained on the same source data.

> If the classifier outputs are independent, then it can be shown that majority voting is the optimal combination rule.
>
> — Page 7, *Ensemble Machine Learning*, 2012

Hard voting is appropriate when the models used in the voting ensemble predict crisp class labels. Soft voting is appropriate when the models used in the voting ensemble predict the probability of class membership. Soft voting can be used for models that do not natively predict a class membership probability, although may require calibration of their probability-like scores prior to being used in the ensemble (e.g. support vector machine, k-nearest neighbors, and decision trees).

- Hard voting is for models that predict class labels.
- Soft voting is for models that predict class membership probabilities.

The voting ensemble is not guaranteed to provide better performance than any single model used in the ensemble. If any given model used in the ensemble performs better than the voting ensemble, that model should probably be used instead of the voting ensemble. This is not always the case. A voting ensemble can offer lower variance in the predictions made over individual models. This can be seen in a lower variance in prediction error for regression tasks. This can also be seen in a lower variance in accuracy for classification tasks. This lower variance may result in a lower mean performance of the ensemble, which might be desirable given the higher stability or confidence of the model. Use a voting ensemble if:

- It results in better performance than any model used in the ensemble.
- It results in a lower variance than any model used in the ensemble.

A voting ensemble is particularly useful for machine learning models that use a stochastic learning algorithm and result in a different final model each time it is trained on the same dataset. One example is neural networks that are fit using stochastic gradient descent. Another useful case for voting ensembles is when combining multiple fits of the same machine learning algorithm with slightly different hyperparameters. Voting ensembles are most effective when:

- Combining multiple fits of a model trained using stochastic learning algorithms.
- Combining multiple fits of a model with different hyperparameters.

A limitation of the voting ensemble is that it treats all models the same, meaning all models contribute equally to the prediction. This is a problem if some models are good in some situations and poor in others.

25.3 Develop Voting Ensembles

The scikit-learn Python machine learning library provides an implementation of voting for machine learning via the `VotingRegressor` and `VotingClassifier` classes. Both models operate the same way and take the same arguments. Using the model requires that you specify a list of estimators that make predictions and are combined in the voting ensemble. A list of base-models is provided via the `estimators` argument. This is a Python list where each element in the list is a tuple with the name of the model and the configured model instance. Each model in the list must have a unique name. For example, below defines two base-models:

```
...
models = [('lr',LogisticRegression()),('svm',SVC())]
ensemble = VotingClassifier(estimators=models)
```

Listing 25.1: Example of defining a voting ensemble for classification.

Each model in the list may also be a `Pipeline`, including any data preparation required by the model prior to fitting the model on the training dataset. For example:

```
...
models = [('lr',LogisticRegression()),('svm',make_pipeline(StandardScaler(),SVC()))]
ensemble = VotingClassifier(estimators=models)
```

Listing 25.2: Example of defining a voting ensemble for classification that includes a pipeline.

When using a voting ensemble for classification, the type of voting, such as hard voting or soft voting, can be specified via the `voting` argument and set to the string `'hard'` (the default) or `'soft'`. For example:

```
...
models = [('lr',LogisticRegression()),('svm',SVC())]
ensemble = VotingClassifier(estimators=models, voting='soft')
```

Listing 25.3: Example of defining a soft voting ensemble for classification.

Now that we are familiar with the voting ensemble API in scikit-learn, let's look at some worked examples.

25.4 Voting Ensemble for Classification

In this section, we will look at using stacking for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=2)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 25.4: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 25.5: Example output from creating the synthetic classification dataset.

Next, we will demonstrate hard voting and soft voting for this dataset.

25.4.1 Hard Voting Ensemble for Classification

We can demonstrate hard voting with a k-nearest neighbor algorithm. We can fit five different versions of the KNN algorithm, each with a different number of neighbors used when making predictions. We will use 1, 3, 5, 7, and 9 neighbors (odd numbers in an attempt to avoid ties). Our expectation is that by combining the predicted class labels predicted by each different KNN model that the hard voting ensemble will achieve a better predictive performance than any standalone model used in the ensemble, on average. We can then create a list of models to evaluate, including each standalone version of the KNN model configurations and the hard voting ensemble. This will help us directly compare each standalone configuration of the KNN model with the ensemble in terms of the distribution of classification accuracy scores. The `get_models()` function below creates the list of models for us to evaluate.

```
# get a list of standalone models to evaluate
def get_models():
  models = dict()
  # define the number of neighbors to consider
  neighbors = [1, 3, 5, 7, 9]
  for n in neighbors:
    key = 'knn' + str(n)
    models[key] = KNeighborsClassifier(n_neighbors=n)
  # define the voting ensemble
  members = [(n,m) for n,m in models.items()]
  models['hard_voting'] = VotingClassifier(estimators=members, voting='hard')
  return models
```

Listing 25.6: Example of a function for defining models for comparison with the hard voting ensemble.

Each model will be evaluated using repeated k-fold cross-validation. The `evaluate_model()` function below takes a model instance and returns as a list of scores from three repeats of stratified 10-fold cross-validation.

```
# compare hard voting to standalone classifiers
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.neighbors import KNeighborsClassifier
from sklearn.ensemble import VotingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=2)
  return X, y

# get a list of standalone models to evaluate
def get_models():
  models = dict()
  # define the number of neighbors to consider
  neighbors = [1, 3, 5, 7, 9]
  for n in neighbors:
```

```
    key = 'knn' + str(n)
    models[key] = KNeighborsClassifier(n_neighbors=n)
  # define the voting ensemble
  members = [(n,m) for n,m in models.items()]
  models['hard_voting'] = VotingClassifier(estimators=members, voting='hard')
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 25.7: Example of comparing the performance of a hard voting ensemble to contributing models.

Running the example first reports the mean and standard deviation accuracy for each model.

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

We can see the hard voting ensemble achieves a better classification accuracy of about 90.2 percent compared to all standalone versions of the model.

```
>knn1 0.873 (0.030)
>knn3 0.889 (0.038)
>knn5 0.895 (0.031)
>knn7 0.899 (0.035)
>knn9 0.900 (0.033)
>hard_voting 0.902 (0.034)
```

Listing 25.8: Example output from comparing the performance of a hard voting ensemble to contributing models.

A box-and-whisker plot is then created comparing the distribution accuracy scores for each model, allowing us to clearly see that hard voting ensemble performing better than all standalone models on average.

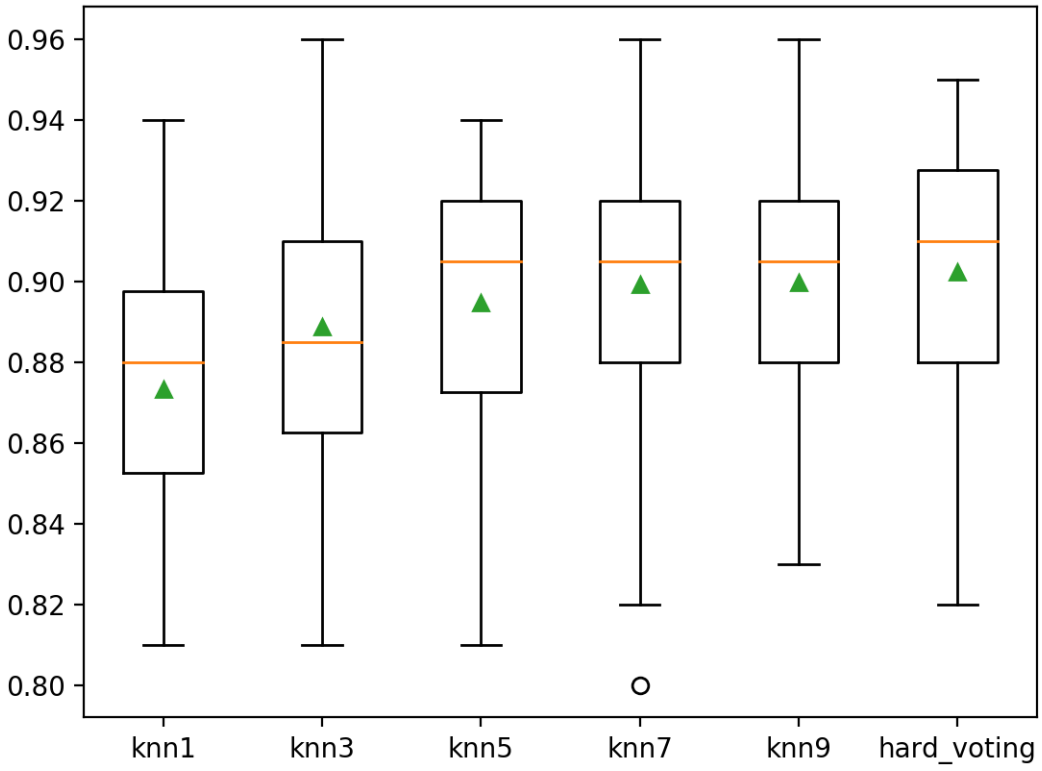

Figure 25.1: Box Plot of Hard Voting Ensemble Compared to Standalone Models for Binary Classification.

If we choose a hard voting ensemble as our final model, we can fit and use it to make predictions on new data just like any other model. First, the hard voting ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with a hard voting ensemble
from sklearn.datasets import make_classification
from sklearn.ensemble import VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=2)
# define the base models
models = list()
neighbors = [1, 3, 5, 7, 9]
for n in neighbors:
  models.append(('knn'+str(n), KNeighborsClassifier(n_neighbors=n)))
# define the hard voting ensemble
```

```
ensemble = VotingClassifier(estimators=models, voting='hard')
# fit the model on all available data
ensemble.fit(X, y)
# make a prediction for one example
row = [5.88891819, 2.64867662, -0.42728226, -1.24988856, -0.00822, -3.57895574, 2.87938412,
    -1.55614691, -0.38168784, 7.50285659, -1.16710354, -5.02492712, -0.46196105,
    -0.64539455, -1.71297469, 0.25987852, -0.193401, -5.52022952, 0.0364453, -1.960039]
# summarize the prediction
yhat = ensemble.predict([row])
print('Predicted Class: %d' % (yhat))
```

Listing 25.9: Example of making a prediction with a hard voting ensemble.

Running the example fits the hard voting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 25.10: Example output from making a prediction with a hard voting ensemble.

25.4.2 Soft Voting Ensemble for Classification

We can demonstrate soft voting with the support vector machine (SVM) algorithm. The SVM algorithm does not natively predict probabilities, although it can be configured to predict probability-like scores by setting the `probability` argument to `True` in the SVC class. We can fit five different versions of the SVM algorithm with a polynomial kernel, each with a different polynomial degree, set via the `degree` argument. We will use degrees 1-5. Our expectation is that by combining the predicted class membership probability scores predicted by each different SVM model that the soft voting ensemble will achieve a better predictive performance than any standalone model used in the ensemble, on average. We can then create a list of models to evaluate, including each standalone version of the SVM model configurations and the soft voting ensemble. This will help us directly compare each standalone configuration of the SVM model with the ensemble in terms of the distribution of classification accuracy scores. The `get_models()` function below creates the list of models for us to evaluate.

```
# get a list of standalone models to evaluate
def get_models():
  models = dict()
  # define the degrees to consider
  for n in range(1,6):
    key = 'svm' + str(n)
    models[key] = SVC(probability=True, kernel='poly', degree=n)
  # define the voting ensemble
  members = [(n,m) for n,m in models.items()]
  models['soft_voting'] = VotingClassifier(estimators=members, voting='soft')
  return models
```

Listing 25.11: Example of a function for defining models for comparison with the soft voting ensemble.

We can evaluate and report model performance using repeated k-fold cross-validation as we did in the previous section. Tying this together, the complete example is listed below.

```
# compare soft voting ensemble to standalone classifiers
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.svm import SVC
from sklearn.ensemble import VotingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=2)
  return X, y

# get a list of standalone models to evaluate
def get_models():
  models = dict()
  # define the degrees to consider
  for n in range(1,6):
    key = 'svm' + str(n)
    models[key] = SVC(probability=True, kernel='poly', degree=n)
  # define the voting ensemble
  members = [(n,m) for n,m in models.items()]
  models['soft_voting'] = VotingClassifier(estimators=members, voting='soft')
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 25.12: Example of comparing the performance of a soft voting ensemble to contributing

models.

Running the example first reports the mean and standard deviation accuracy for each model.

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

We can see the soft voting ensemble achieves a better classification accuracy of about 92.4 percent compared to all standalone versions of the model.

```
>svm1 0.855 (0.035)
>svm2 0.859 (0.034)
>svm3 0.890 (0.035)
>svm4 0.808 (0.037)
>svm5 0.850 (0.037)
>soft_voting 0.924 (0.028)
```

Listing 25.13: Example output from comparing the performance of a soft voting ensemble to contributing models.

A box-and-whisker plot is then created comparing the distribution accuracy scores for each model, allowing us to clearly see that soft voting ensemble performing better than all standalone models on average.

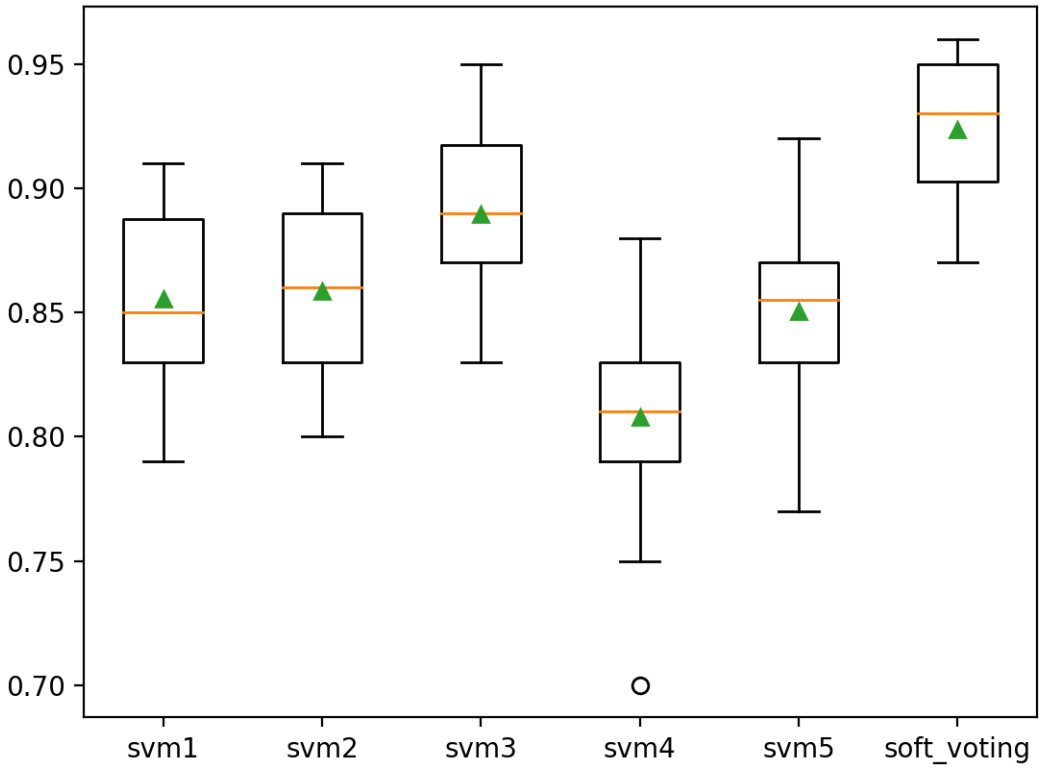

Figure 25.2: Box Plot of Soft Voting Ensemble Compared to Standalone Models for Binary Classification.

If we choose a soft voting ensemble as our final model, we can fit and use it to make predictions on new data just like any other model. First, the soft voting ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with a soft voting ensemble
from sklearn.datasets import make_classification
from sklearn.ensemble import VotingClassifier
from sklearn.svm import SVC
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=2)
# define the base models
models = list()
for n in range(1,6):
  models.append(('svm'+str(n), SVC(probability=True, kernel='poly', degree=n)))
# define the soft voting ensemble
ensemble = VotingClassifier(estimators=models, voting='soft')
# fit the model on all available data
ensemble.fit(X, y)
# make a prediction for one example
row = [5.88891819, 2.64867662, -0.42728226, -1.24988856, -0.00822, -3.57895574, 2.87938412,
```

```
    -1.55614691, -0.38168784, 7.50285659, -1.16710354, -5.02492712, -0.46196105,
    -0.64539455, -1.71297469, 0.25987852, -0.193401, -5.52022952, 0.0364453, -1.960039]
yhat = ensemble.predict([row])
# summarize prediction
print('Predicted Class: %d' % (yhat))
```

Listing 25.14: Example of making a prediction with a soft voting ensemble.

Running the example fits the soft voting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 1
```

Listing 25.15: Example output from making a prediction with a soft voting ensemble.

25.5 Voting Ensemble for Regression

In this section, we will look at using voting for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 25.16: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 25.17: Example output from creating the synthetic regression dataset.

We can demonstrate ensemble voting for regression with a decision tree algorithm, sometimes referred to as a classification and regression tree (CART) algorithm. We can fit five different versions of the CART algorithm, each with a different maximum depth of the decision tree, set via the `max_depth` argument. We will use depths of 1-5. Our expectation is that by combining the values predicted by each different CART model that the voting ensemble will achieve a better predictive performance than any standalone model used in the ensemble, on average. We can then create a list of models to evaluate, including each standalone version of the CART model configurations and the soft voting ensemble. This will help us directly compare each standalone configuration of the CART model with the ensemble in terms of the distribution of error scores. The `get_models()` function below creates the list of models for us to evaluate.

```
# get a list of standalone models to evaluate
def get_models():
  models = dict()
  # define the tree depths to consider
```

```
  for n in range(1,6):
    key = 'cart' + str(n)
    models[key] = DecisionTreeRegressor(max_depth=n)
  # define the voting ensemble
  members = [(n,m) for n,m in models.items()]
  models['voting'] = VotingRegressor(estimators=members)
  return models
```

Listing 25.18: Example of a function for defining models for comparison with the voting ensemble for regression.

We can evaluate and report model performance using repeated k-fold cross-validation as we did in the previous section. Models are evaluated using mean absolute error (MAE). Tying this together, the complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# compare voting ensemble to each standalone models for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
      random_state=1)
  return X, y

# get a list of standalone models to evaluate
def get_models():
  models = dict()
  # define the tree depths to consider
  for n in range(1,6):
    key = 'cart' + str(n)
    models[key] = DecisionTreeRegressor(max_depth=n)
  # define the voting ensemble
  members = [(n,m) for n,m in models.items()]
  models['voting'] = VotingRegressor(estimators=members)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
```

```
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 25.19: Example of comparing the performance of a voting ensemble to contributing models for regression.

Running the example first reports the mean and standard deviation accuracy for each model.

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

We can see the voting ensemble achieves a better mean absolute error of about -136.338, which is larger (better) compared to all standalone versions of the model.

```
>cart1 -161.519 (11.414)
>cart2 -152.596 (11.271)
>cart3 -142.378 (10.900)
>cart4 -140.086 (12.469)
>cart5 -137.641 (12.240)
>voting -136.338 (11.242)
```

Listing 25.20: Example output from comparing the performance of a voting ensemble to contributing models for regression.

A box-and-whisker plot is then created comparing the distribution negative MAE scores for each model, allowing us to clearly see that voting ensemble performed better than all standalone models on average.

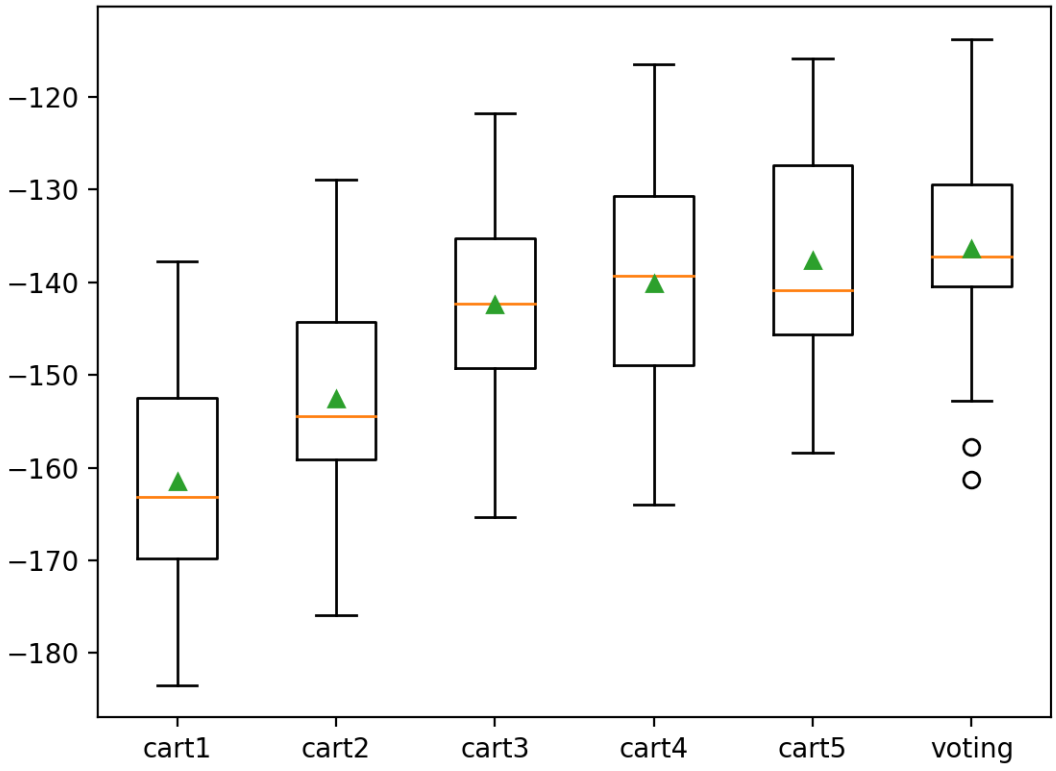

Figure 25.3: Box Plot of Voting Ensemble Compared to Standalone Models for Regression.

If we choose a voting ensemble as our final model, we can fit and use it to make predictions on new data just like any other model. First, the voting ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with a voting ensemble
from sklearn.datasets import make_regression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=1)
# define the base models
models = list()
for n in range(1,6):
  models.append(('cart'+str(n), DecisionTreeRegressor(max_depth=n)))
# define the voting ensemble
ensemble = VotingRegressor(estimators=models)
# fit the model on all available data
ensemble.fit(X, y)
# make a prediction for one example
row = [0.59332206, -0.56637507, 1.34808718, -0.57054047, -0.72480487, 1.05648449,
    0.77744852, 0.07361796, 0.88398267, 2.02843157, 1.01902732, 0.11227799, 0.94218853,
```

```
    0.26741783, 0.91458143, -0.72759572, 1.08842814, -0.61450942, -0.69387293, 1.69169009]
yhat = ensemble.predict([row])
# summarize prediction
print('Predicted Value: %.3f' % (yhat))
```

Listing 25.21: Example of making a prediction with a voting ensemble for regression.

Running the example fits the voting ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Value: 141.319
```

Listing 25.22: Example output from making a prediction with a voting ensemble for regression.

25.6 Further Reading

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

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5
- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2NJT1L8

APIs

- Ensemble methods scikit-learn API.
  https://scikit-learn.org/stable/modules/ensemble.html
- `sklearn.ensemble.VotingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html
- `sklearn.ensemble.VotingRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingRegressor.html

25.7 Summary

In this tutorial, you discovered how to create voting ensembles for machine learning algorithms in Python. Specifically, you learned:

- A voting ensemble involves summing the predictions made by classification models or averaging the predictions made by regression models.

- How voting ensembles work, when to use voting ensembles, and the limitations of the approach.
- How to implement a hard voting ensemble and soft voting ensembles for classification predictive modeling.

Next

In the next section, we will take a closer look at an extension to voting called the weighted average ensemble.

Chapter 26

Weighted Average Ensemble

Weighted average ensembles assume that some models in the ensemble have more skill than others and give them more contribution when making predictions. The weighted average or weighted sum ensemble is an extension to voting ensembles that assume all models are equally skillful and make the same proportional contribution to predictions made by the ensemble. Each model is assigned a fixed weight that is multiplied by the prediction made by the model and used in the sum or average prediction calculation. The challenge of this type of ensemble is how to calculate, assign, or search for model weights that result in performance that is better than any contributing model and an ensemble that uses equal model weights. In this tutorial, you will discover how to develop Weighted Average Ensembles for classification and regression. After completing this tutorial, you will know:

- Weighted Average Ensembles are an extension to voting ensembles where model votes are proportional to model performance.
- How to develop weighted average ensembles using the voting ensemble from scikit-learn.
- How to evaluate the Weighted Average Ensembles for classification and regression and confirm the models are skillful.

Let's get started.

26.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Weighted Average Ensemble
2. Develop a Weighted Average Ensemble
3. Weighted Average Ensemble for Classification
4. Weighted Average Ensemble for Regression

26.2 Weighted Average Ensemble

Weighted average or weighted sum ensemble is an ensemble machine learning approach that combines the predictions from multiple models, where the contribution of each model is weighted proportionally to its capability or skill. The weighted average ensemble is related to the voting ensemble. Voting ensembles are composed of multiple machine learning models where the predictions from each model are averaged directly. For regression, this involves calculating the arithmetic mean of the predictions made by ensemble members. For classification, this may involve calculating the statistical mode (most common class label) or similar voting scheme or summing the probabilities predicted for each class and selecting the class with the largest summed probability.

A limitation of the voting ensemble technique is that it assumes that all models in the ensemble are equally effective. This may not be the case as some models may be better than others, especially if different machine learning algorithms are used to train each model ensemble member. An alternative to voting is to assume that ensemble members are not all equally capable and instead some models are better than others and should be given more votes or more of a seat when making a prediction. This provides the motivation for the weighted sum or weighted average ensemble method. In regression, an average prediction is calculated using the arithmetic mean, such as the sum of the predictions divided by the total predictions made. For example, if an ensemble had three ensemble members, the reductions may be:

- **Model 1**: 97.2
- **Model 2**: 100.0
- **Model 3**: 95.8

The mean prediction would be calculated as follows:

$$\begin{aligned} \text{yhat} &= \frac{97.2 + 100.0 + 95.8}{3} \\ &= \frac{293}{3} \\ &= 97.666 \end{aligned} \tag{26.1}$$

A weighted average prediction involves first assigning a fixed weight coefficient to each ensemble member. This could be a floating-point value between 0 and 1, representing a percentage of the weight. It could also be an integer starting at 1, representing the number of votes to give each model. For example, we may have the fixed weights of 0.84, 0.87, 0.75 for the ensemble member. These weights can be used to calculate the weighted average by multiplying each prediction by the model's weight to give a weighted sum, then dividing the value as before. For example:

$$\begin{aligned} \text{yhat} &= \frac{(97.2 \times 0.84) + (100.0 \times 0.87) + (95.8 \times 0.75)}{3} \\ &= \frac{81.648 + 87 + 71.85}{3} \\ &= \frac{240.498}{3} \\ &= 80.166 \end{aligned} \tag{26.2}$$

We can see that as long as the scores have the same scale, and the weights have the same scale and are maximizing (meaning that larger weights are better), the weighted sum results in a sensible value, and in turn, the weighted average is also sensible, meaning the scale of the outcome matches the scale of the scores. This same approach can be used to calculate the weighted sum of votes for each crisp class label or the weighted sum of probabilities for each class label on a classification problem. The challenging aspect of using a weighted average ensemble is how to choose the relative weighting for each ensemble member.

There are many approaches that can be used. For example, the weights may be chosen based on the skill of each model, such as the classification accuracy or negative error, where large weights mean a better-performing model. Performance may be calculated on the dataset used for training or a holdout dataset, the latter of which may be more relevant. The scores of each model can be used directly or converted into a different value, such as the relative ranking for each model. Another approach might be to use a search algorithm to test different combinations of weights. Now that we are familiar with the weighted average ensemble method, let's look at how to develop and evaluate them.

26.3 Develop a Weighted Average Ensemble

In this section, we will develop, evaluate, and use weighted average or weighted sum ensemble models. We can implement weighted average ensembles manually, although this is not required as we can use the voting ensemble in the scikit-learn library to achieve the desired effect (introduced in Chapter 25). Specifically, the `VotingRegressor` and `VotingClassifier` classes can be used for regression and classification respectively and both provide a `weights` argument that specifies the relative contribution of each ensemble member when making a prediction. A list of base-models is provided via the `estimators` argument. This is a Python list where each element in the list is a tuple with the name of the model and the configured model instance. Each model in the list must have a unique name. For example, we can define a weighted average ensemble for classification with two ensemble members as follows:

```
...
# define the models in the ensemble
models = [('lr',LogisticRegression()),('svm',SVC())]
# define the weight of each model in the ensemble
weights = [0.7, 0.9]
# create a weighted sum ensemble
ensemble = VotingClassifier(estimators=models, weights=weights)
```

Listing 26.1: Example of defining a weighted average ensemble via hard voting.

Additionally, the voting ensemble for classification provides the `voting` argument that supports both hard voting (`'hard'`) for combining crisp class labels and soft voting (`'soft'`) for combining class probabilities when calculating the weighted sum for prediction; for example:

```
...
# define the models in the ensemble
models = [('lr',LogisticRegression()),('svm',SVC())]
# define the weight of each model in the ensemble
weights = [0.7, 0.9]
# create a weighted sum ensemble
ensemble = VotingClassifier(estimators=models, weights=weights, voting='soft')
```

Listing 26.2: Example of defining a weighted average ensemble via soft voting.

Soft voting is generally preferred if the contributing models support predicting class probabilities, as it often results in better performance. The same holds for the weighted sum of predicted probabilities. Now that we are familiar with how to use the voting ensemble API to develop weighted average ensembles, let's look at some worked examples.

26.4 Weighted Average Ensemble for Classification

In this section, we will look at using Weighted Average Ensemble for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 10,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 26.3: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(10000, 20) (10000,)
```

Listing 26.4: Example output from creating the synthetic classification dataset.

Next, we can evaluate a Weighted Average Ensemble algorithm on this dataset. First, we will split the dataset into train and test sets with a 50-50 split. We will then split the full training set into a subset for training the models and a subset for validation.

```
...
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.50,
    random_state=1)
# split the full train set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
```

Listing 26.5: Example of splitting the dataset into train, validation, and test sets.

Next, we will define a function to create a list of models to use in the ensemble. In this case, we will use a diverse collection of classification models, including logistic regression, a decision tree, and naive Bayes.

```
# get a list of base-models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('bayes', GaussianNB()))
```

```
  return models
```

Listing 26.6: Example of defining a function for creating the models to use in the ensemble.

Next, we need to weigh each ensemble member. In this case, we will use the performance of each ensemble model on the training dataset as the relative weighting of the model when making predictions. Performance will be calculated using classification accuracy as a percentage of correct predictions between 0 and 1, with larger values meaning a better model, and in turn, more contribution to the prediction. Each ensemble model will first be fit on the training set, then evaluated on the validation set. The accuracy on the validation set will be used as the model weighting. The `evaluate_models()` function below implements this, returning the performance of each model.

```
# evaluate each base-model
def evaluate_models(models, X_train, X_val, y_train, y_val):
  # fit and evaluate the models
  scores = list()
  for _, model in models:
    # fit the model
    model.fit(X_train, y_train)
    # evaluate the model
    yhat = model.predict(X_val)
    acc = accuracy_score(y_val, yhat)
    # store the performance
    scores.append(acc)
    # report model performance
  return scores
```

Listing 26.7: Example of defining a function evaluating a list of models.

We can then call this function to get the scores and use them as a weighting for the ensemble.

```
...
# fit and evaluate each model
scores = evaluate_models(models, X_train, X_val, y_train, y_val)
# create the ensemble
ensemble = VotingClassifier(estimators=models, voting='soft', weights=scores)
```

Listing 26.8: Example of creating the weighted average ensemble for classification.

We can then fit the ensemble on the full training dataset and evaluate it on the holdout test set.

```
...
# fit the ensemble on the training dataset
ensemble.fit(X_train, y_train)
# make predictions on test set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Weighted Avg Accuracy: %.3f' % (score*100))
```

Listing 26.9: Example of evaluating the weighted average ensemble for regression.

Tying this together, the complete example is listed below.

```
# evaluate a weighted average ensemble for classification
```

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier

# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('bayes', GaussianNB()))
  return models

# evaluate each base model
def evaluate_models(models, X_train, X_val, y_train, y_val):
  # fit and evaluate the models
  scores = list()
  for _, model in models:
    # fit the model
    model.fit(X_train, y_train)
    # evaluate the model
    yhat = model.predict(X_val)
    acc = accuracy_score(y_val, yhat)
    # store the performance
    scores.append(acc)
    # report model performance
  return scores

# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.50,
    random_state=1)
# split the full train set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# create the base models
models = get_models()
# fit and evaluate each model
scores = evaluate_models(models, X_train, X_val, y_train, y_val)
print(scores)
# create the ensemble
ensemble = VotingClassifier(estimators=models, voting='soft', weights=scores)
# fit the ensemble on the training dataset
ensemble.fit(X_train_full, y_train_full)
# make predictions on test set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Weighted Avg Accuracy: %.3f' % (score*100))
```

Listing 26.10: Example of evaluating the weighted average ensemble.

Running the example first evaluates each standalone model and reports the accuracy scores that will be used as model weights. Finally, the weighted average ensemble is fit and evaluated on the test reporting the performance.

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

In this case, we can see that the voting ensemble achieved a classification accuracy of about 90.960 percent.

```
[0.8896969696969697, 0.8575757575757575, 0.8812121212121212]
Weighted Avg Accuracy: 90.960
```

Listing 26.11: Example output from evaluating the weighted average ensemble.

Our expectation is that the ensemble will perform better than any of the contributing ensemble members. The problem is the accuracy scores for the models used as weightings cannot be directly compared to the performance of the ensemble because the members were evaluated on a subset of training and the ensemble was evaluated on the test dataset. We can update the example and add an evaluation of each standalone model for comparison.

```
...
# evaluate each standalone model
scores = evaluate_models(models, X_train_full, X_test, y_train_full, y_test)
for i in range(len(models)):
  print('>%s: %.3f' % (models[i][0], scores[i]*100))
```

Listing 26.12: Example of evaluating each standalone model that contributes to the weighted average ensemble.

We also expect the weighted average ensemble to perform better than an equally weighted voting ensemble. This can also be checked by explicitly evaluating the voting ensemble.

```
...
# evaluate equal weighting
ensemble = VotingClassifier(estimators=models, voting='soft')
ensemble.fit(X_train_full, y_train_full)
yhat = ensemble.predict(X_test)
score = accuracy_score(y_test, yhat)
print('Voting Accuracy: %.3f' % (score*100))
```

Listing 26.13: Example of evaluating an equally weighted average ensemble.

Tying this together, the complete example is listed below.

```
# evaluate a weighted average ensemble for classification compared to base model
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier

# get a list of base models
```

```
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('bayes', GaussianNB()))
  return models

# evaluate each base model
def evaluate_models(models, X_train, X_val, y_train, y_val):
  # fit and evaluate the models
  scores = list()
  for _, model in models:
    # fit the model
    model.fit(X_train, y_train)
    # evaluate the model
    yhat = model.predict(X_val)
    acc = accuracy_score(y_val, yhat)
    # store the performance
    scores.append(acc)
    # report model performance
  return scores

# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.50,
    random_state=1)
# split the full train set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# create the base models
models = get_models()
# fit and evaluate each model
scores = evaluate_models(models, X_train, X_val, y_train, y_val)
print(scores)
# create the ensemble
ensemble = VotingClassifier(estimators=models, voting='soft', weights=scores)
# fit the ensemble on the training dataset
ensemble.fit(X_train_full, y_train_full)
# make predictions on test set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Weighted Avg Accuracy: %.3f' % (score*100))
# evaluate each standalone model
scores = evaluate_models(models, X_train_full, X_test, y_train_full, y_test)
for i in range(len(models)):
  print('>%s: %.3f' % (models[i][0], scores[i]*100))
# evaluate equal weighting
ensemble = VotingClassifier(estimators=models, voting='soft')
ensemble.fit(X_train_full, y_train_full)
yhat = ensemble.predict(X_test)
score = accuracy_score(y_test, yhat)
print('Voting Accuracy: %.3f' % (score*100))
```

Listing 26.14: Example of comparing the weighted average ensemble to the standalone models and voting ensemble.

Running the example first prepares and evaluates the weighted average ensemble as before, then reports the performance of each contributing model evaluated in isolation, and finally the voting ensemble that uses an equal weighting for the contributing models.

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

In this case, we can see that the weighted average ensemble performs better than any contributing ensemble member. We can also see that an equal weighting ensemble (voting) achieved an accuracy of about 90.620, which is less than the weighted ensemble that achieved the slightly higher 90.760 percent accuracy.

```
[0.8896969696969697, 0.8703030303030304, 0.8812121212121212]
Weighted Avg Accuracy: 90.760
>lr: 87.800
>cart: 88.180
>bayes: 87.300
Voting Accuracy: 90.620
```

Listing 26.15: Example output from comparing the weighted average ensemble to the standalone models and voting ensemble.

Next, let's take a look at how to develop and evaluate a weighted average ensemble for regression.

26.5 Weighted Average Ensemble for Regression

In this section, we will look at using Weighted Average Ensemble for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=10000, n_features=20, n_informative=10, noise=0.3,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 26.16: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(10000, 20) (10000,)
```

Listing 26.17: Example output from creating the synthetic classification dataset.

Next, we can evaluate a Weighted Average Ensemble model on this dataset. First, we can split the dataset into train and test sets, then further split the training set into train and validation sets so that we can estimate the performance of each contributing model.

```
...
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.50,
    random_state=1)
# split the full train set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
```

Listing 26.18: Example of splitting the dataset into train, validation, and test sets.

We can define the list of models to use in the ensemble. In this case, we will use k-nearest neighbors, decision tree, and support vector regression.

```
# get a list of base-models
def get_models():
  models = list()
  models.append(('knn', KNeighborsRegressor()))
  models.append(('cart', DecisionTreeRegressor()))
  models.append(('svm', SVR()))
  return models
```

Listing 26.19: Example of defining a function for creating the models to use in the ensemble.

Next, we can update the `evaluate_models()` function to calculate the mean absolute error (MAE) for each ensemble member on a hold out validation dataset. We will use the negative MAE scores (e.g. maximizing) as a weight where smaller error values closer to zero indicate a better performing model.

```
# evaluate each base-model
def evaluate_models(models, X_train, X_val, y_train, y_val):
  # fit and evaluate the models
  scores = list()
  for _, model in models:
    # fit the model
    model.fit(X_train, y_train)
    # evaluate the model
    yhat = model.predict(X_val)
    mae = mean_absolute_error(y_val, yhat)
    # store the performance
    scores.append(-mae)
    # report model performance
  return scores
```

Listing 26.20: Example of defining a function evaluating a list of models.

We can then call this function to get the scores and use them to define the weighted average ensemble for regression.

```
...
# fit and evaluate each model
scores = evaluate_models(models, X_train, X_val, y_train, y_val)
print(scores)
# create the ensemble
```

```
ensemble = VotingRegressor(estimators=models, weights=scores)
```

Listing 26.21: Example of creating the weighted average ensemble for regression.

We can then fit the ensemble on the entire training dataset and evaluate the performance on the holdout test dataset.

```
...
# fit the ensemble on the training dataset
ensemble.fit(X_train_full, y_train_full)
# make predictions on test set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = mean_absolute_error(y_test, yhat)
print('Weighted Avg MAE: %.3f' % (score))
```

Listing 26.22: Example of evaluating the weighted average ensemble for regression.

We expect the ensemble to perform better than any contributing ensemble member, and this can be checked directly by evaluating each member model on the full train and test sets independently.

```
...
# evaluate each standalone model
scores = evaluate_models(models, X_train_full, X_test, y_train_full, y_test)
for i in range(len(models)):
  print('>%s: %.3f' % (models[i][0], scores[i]))
```

Listing 26.23: Example of evaluating each standalone model that contributes to the weighted average ensemble.

Finally, we also expect the weighted average ensemble to perform better than the same ensemble with an equal weighting. This too can be confirmed.

```
...
# evaluate equal weighting
ensemble = VotingRegressor(estimators=models)
ensemble.fit(X_train_full, y_train_full)
yhat = ensemble.predict(X_test)
score = mean_absolute_error(y_test, yhat)
print('Voting MAE: %.3f' % (score))
```

Listing 26.24: Example of evaluating an equally weighted average ensemble.

Tying this together, the complete example of evaluating a weighted average ensemble for regression is listed below.

```
# evaluate a weighted average ensemble for regression
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import VotingRegressor

# get a list of base models
def get_models():
```

```
  models = list()
  models.append(('knn', KNeighborsRegressor()))
  models.append(('cart', DecisionTreeRegressor()))
  models.append(('svm', SVR()))
  return models

# evaluate each base model
def evaluate_models(models, X_train, X_val, y_train, y_val):
  # fit and evaluate the models
  scores = list()
  for _, model in models:
    # fit the model
    model.fit(X_train, y_train)
    # evaluate the model
    yhat = model.predict(X_val)
    mae = mean_absolute_error(y_val, yhat)
    # store the performance
    scores.append(-mae)
    # report model performance
  return scores

# define dataset
X, y = make_regression(n_samples=10000, n_features=20, n_informative=10, noise=0.3,
    random_state=7)
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.50,
    random_state=1)
# split the full train set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# create the base models
models = get_models()
# fit and evaluate each model
scores = evaluate_models(models, X_train, X_val, y_train, y_val)
print(scores)
# create the ensemble
ensemble = VotingRegressor(estimators=models, weights=scores)
# fit the ensemble on the training dataset
ensemble.fit(X_train_full, y_train_full)
# make predictions on test set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = mean_absolute_error(y_test, yhat)
print('Weighted Avg MAE: %.3f' % (score))
# evaluate each standalone model
scores = evaluate_models(models, X_train_full, X_test, y_train_full, y_test)
for i in range(len(models)):
  print('>%s: %.3f' % (models[i][0], scores[i]))
# evaluate equal weighting
ensemble = VotingRegressor(estimators=models)
ensemble.fit(X_train_full, y_train_full)
yhat = ensemble.predict(X_test)
score = mean_absolute_error(y_test, yhat)
print('Voting MAE: %.3f' % (score))
```

Listing 26.25: Example of comparing the weighted average ensemble to the standalone models

and voting ensemble for regression.

Running the example first reports the MAE of each ensemble member that will be used as scores, followed by the performance of the weighted average ensemble. Finally, the performance of each independent model is reported along with the performance of an ensemble with equal weight.

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

In this case, we can see that the weighted average ensemble achieved a mean absolute error of about 105.158, which is worse (large error) than the standalone kNN model that achieved an error of about 100.169. We can also see that the voting ensemble that assumes an equal weight for each model also performs better than the weighted average ensemble with an error of about 102.706.

```
[0.8896969696969697, 0.8703030303030304, 0.8812121212121212]
[-101.97000126284476, -142.26014983127837, -153.9765827528269]
Weighted Avg MAE: 105.158
>knn: -100.169
>cart: -134.487
>svm: -138.195
Voting MAE: 102.706
```

Listing 26.26: Example output from comparing the weighted average ensemble to the standalone models and voting ensemble for regression.

The worse-than-expected performance for the weighted average ensemble might be related to the choice of how models were weighted. An alternate strategy for weighting is to use a ranking to indicate the number of votes that each ensemble has in the weighted average. For example, the worst-performing model has 1 vote the second-worst 2 votes and the best model 3 votes, in the case of three ensemble members. This can be achieved using the `argsort()` NumPy function.

The argsort function returns the indexes of the values in an array if they were sorted. So, if we had the array [300, 100, 200], the index of the smallest value is 1, the index of the next largest value is 2, and the index of the next largest value is 0. Therefore, the argsort of [300, 100, 200] is [1, 2, 0]. We can then argsort the result of the argsort to give a ranking of the data in the original array. To see how, an argsort of [1, 2, 0] would indicate that index 2 is the smallest value, followed by index 0 and ending with index 1. Therefore, the argsort of [1, 2, 0] is [2, 0, 1]. Put another way, the argsort of the argsort of [300, 100, 200] is [2, 0, 1], which is the relative ranking of each value in the array if values were sorted in ascending order. That is:

- 300: Has rank 2
- 100: Has rank 0
- 200: Has rank 1

We can make this clear with a small example, listed below.

```
# demonstrate argsort
from numpy import argsort
# data
x = [300, 100, 200]
print(x)
# argsort of data
print(argsort(x))
# arg sort of argsort of data
print(argsort(argsort(x)))
```

Listing 26.27: Example of demonstrating the argsort function with positive scores.

Running the example first reports the raw data, then the argsort of the raw data and the argsort of the argsort of the raw data. The results match our manual calculation.

```
[300, 100, 200]
[1 2 0]
[2 0 1]
```

Listing 26.28: Example output from demonstrating the argsort function with positive scores.

We can use the argsort of the argsort of the model scores to calculate a relative ranking of each ensemble member. If negative mean absolute errors are sorted in ascending order, then the best model would have the largest negative error, and in turn, the highest rank. The worst performing model would have the smallest negative error, and in turn, the lowest rank. Again, we can confirm this with a worked example.

```
# demonstrate argsort with negative scores
from numpy import argsort
# data
x = [-10, -100, -80]
print(x)
# argsort of data
print(argsort(x))
# arg sort of argsort of data
print(argsort(argsort(x)))
```

Listing 26.29: Example of demonstrating the argsort function with negative scores.

Running the example, we can see that the first model has the best score (-10) and the second model has the worst score (-100). The argsort of the argsort of the scores shows that the best model gets the highest rank (most votes) with a value of 2 and the worst model gets the lowest rank (least votes) with a value of 0.

```
[-10, -100, -80]
[1 2 0]
[2 0 1]
```

Listing 26.30: Example output from demonstrating the argsort function with negative scores.

In practice, we don't want any model to have zero votes because it would be excluded from the ensemble. Therefore, we can add 1 to all rankings. After calculating the scores, we can calculate the argsort of the argsort of the model scores to give the rankings. Then use the model rankings as the model weights for the weighted average ensemble.

```
...
# fit and evaluate each model
scores = evaluate_models(models, X_train, X_val, y_train, y_val)
print(scores)
ranking = 1 + argsort(argsort(scores))
print(ranking)
# create the ensemble
ensemble = VotingRegressor(estimators=models, weights=ranking)
```

Listing 26.31: Example of defining the weighted average ensemble using model ranking as a weight.

Tying this together, the complete example of a weighted average ensemble for regression with model ranking used as model weighs is listed below.

```
# evaluate a weighted average ensemble for regression with rankings for model weights
from numpy import argsort
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import VotingRegressor

# get a list of base models
def get_models():
  models = list()
  models.append(('knn', KNeighborsRegressor()))
  models.append(('cart', DecisionTreeRegressor()))
  models.append(('svm', SVR()))
  return models

# evaluate each base model
def evaluate_models(models, X_train, X_val, y_train, y_val):
  # fit and evaluate the models
  scores = list()
  for _, model in models:
    # fit the model
    model.fit(X_train, y_train)
    # evaluate the model
    yhat = model.predict(X_val)
    mae = mean_absolute_error(y_val, yhat)
    # store the performance
    scores.append(-mae)
    # report model performance
  return scores

# define dataset
X, y = make_regression(n_samples=10000, n_features=20, n_informative=10, noise=0.3,
    random_state=7)
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.50,
    random_state=1)
# split the full train set into train and validation sets
```

```
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# create the base models
models = get_models()
# fit and evaluate each model
scores = evaluate_models(models, X_train, X_val, y_train, y_val)
print(scores)
ranking = 1 + argsort(argsort(scores))
print(ranking)
# create the ensemble
ensemble = VotingRegressor(estimators=models, weights=ranking)
# fit the ensemble on the training dataset
ensemble.fit(X_train_full, y_train_full)
# make predictions on test set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = mean_absolute_error(y_test, yhat)
print('Weighted Avg MAE: %.3f' % (score))
# evaluate each standalone model
scores = evaluate_models(models, X_train_full, X_test, y_train_full, y_test)
for i in range(len(models)):
  print('>%s: %.3f' % (models[i][0], scores[i]))
# evaluate equal weighting
ensemble = VotingRegressor(estimators=models)
ensemble.fit(X_train_full, y_train_full)
yhat = ensemble.predict(X_test)
score = mean_absolute_error(y_test, yhat)
print('Voting MAE: %.3f' % (score))
```

Listing 26.32: Example of comparing the weighted average ensemble via model rankings to the standalone models and voting ensemble for regression.

Running the example first scores each model, then converts the scores into rankings. The weighted average ensemble using ranking is then evaluated and compared to the performance of each standalone model and the ensemble with equally weighted models.

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

In this case, we can see that the ranking was performed as expected, with the best-performing member kNN with a score of 101 is assigned the rank of 3, and the other models are ranked accordingly. We can see that the weighted average ensemble achieved the MAE of about 96.692, which is better than any individual model and the unweighted voting ensemble. This highlights the importance of exploring alternative approaches for selecting model weights in the ensemble.

```
[-101.97000126284476, -141.51998518020065, -153.9765827528269]
[3 2 1]
Weighted Avg MAE: 96.692
>knn: -100.169
>cart: -132.976
>svm: -138.195
Voting MAE: 102.832
```

Listing 26.33: Example output from comparing the weighted average ensemble via model rankings to the standalone models and voting ensemble for regression.

26.6 Further Reading

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

APIs

- `numpy.argsort` API.
  https://numpy.org/doc/stable/reference/generated/numpy.argsort.html
- `sklearn.ensemble.VotingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html
- `sklearn.ensemble.VotingRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingRegressor.html

Articles

- Weighted arithmetic mean, Wikipedia.
  https://en.wikipedia.org/wiki/Weighted_arithmetic_mean
- Ensemble averaging (machine learning), Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_averaging_(machine_learning)

26.7 Summary

In this tutorial, you discovered how to develop Weighted Average Ensembles for classification and regression. Specifically, you learned:

- Weighted Average Ensembles are an extension to voting ensembles where model votes are proportional to model performance.
- How to develop weighted average ensembles using the voting ensemble from scikit-learn.
- How to evaluate the Weighted Average Ensembles for classification and regression and confirm the models are skillful.

Next

In the next section, we will take a closer look at how to incrementally grow or prune ensemble members from an ensemble referred to as ensemble member selection.

Chapter 27

Ensemble Member Selection

Ensemble member selection refers to algorithms that optimize the composition of an ensemble. This may involve growing an ensemble from available models or pruning members from a fully defined ensemble. The goal is often to reduce the model or computational complexity of an ensemble with little or no effect on the performance of an ensemble, and in some cases find a combination of ensemble members that results in better performance than blindly using all contributing models directly. In this tutorial, you will discover how to develop ensemble selection algorithms from scratch. After completing this tutorial, you will know:

- Ensemble selection involves choosing a subset of ensemble members that results in lower complexity than using all members and sometimes better performance.
- How to develop and evaluate a greedy ensemble pruning algorithm for classification.
- How to develop and evaluate an algorithm for greedily growing an ensemble from scratch.

Let's get started.

27.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Ensemble Member Selection
2. Baseline Models and Voting
3. Ensemble Pruning Example
4. Ensemble Growing Example

27.2 Ensemble Member Selection

Voting and stacking ensembles typically combine the predictions from a heterogeneous group of model types. Although the ensemble may have a large number of ensemble members, it is hard to know that the best combination of members is being used by the ensemble. For example, instead of simply using all members, it is possible that better results could be achieved by

adding one more different model type or removing one or more models. This can be addressed using a weighted average ensemble and using an optimization algorithm to find an appropriate weighting for each member, allowing some members to have a zero weight, which effectively removes them from the ensemble. The problem with a weighted average ensemble is that all models remain part of the ensemble, perhaps requiring an ensemble of greater complexity than is required to be developed and maintained.

An alternative approach is to optimize the composition of the ensemble itself. The general approach of automatically choosing or optimizing the members of ensembles is referred to as ensemble selection. Two common approaches include ensemble growing and ensemble pruning.

- **Ensemble Growing**: Add members to the ensemble until no further improvement is observed.
- **Ensemble Pruning**: Remove members from the ensemble until no further improvement is observed.

Ensemble growing is a technique where the model starts with no members and involves adding new members until no further improvement is observed. This could be performed in a greedy manner where members are added one at a time only if they result in an improvement in model performance. Ensemble pruning is a technique where the model starts with all possible members that are being considered and removes members from the ensemble until no further improvement is observed. This could be performed in a greedy manner where members are removed one at a time and only if their removal results in a lift in the performance of the overall ensemble.

> Given a set of trained individual learners, rather than combining all of them, ensemble pruning tries to select a subset of individual learners to comprise the ensemble.
>
> — Page 119, *Ensemble Methods: Foundations and Algorithms*, 2012.

An advantage of ensemble pruning and growing is that it may result in an ensemble with a smaller size (lower complexity) and/or an ensemble with better predictive performance. Sometimes a small drop in performance is desirable if it can be achieved in a large drop in model complexity and resulting maintenance burden. Alternately, on some projects, predictive skill is more important than all other concerns, and ensemble selection provides one more strategy to try and get the most out of the contributing models.

> There are two main reasons for reducing the ensemble size: a) Reducing computational overhead: Smaller ensembles require less computational overhead and b) Improving Accuracy: Some members in the ensemble may reduce the predictive performance of the whole.
>
> — Page 119, *Pattern Classification Using Ensemble Methods*, 2010.

Ensemble pruning might be preferred for computational efficiency reasons in cases where a small number of ensemble members are expected to perform better, whereas ensemble growing would be more efficient in cases where a large number of ensemble members may be expected to perform better. Simple greedy ensemble growing and pruning have a lot in common with

stepwise feature selection techniques, such as those used in regression (e.g. so-called stepwise regression). More sophisticated techniques may be used, such as selecting members for addition to or removal from the ensemble based on their standalone performance on the dataset, or even through the use of a global search procedure that attempts to find a combination of ensemble members that results in the best overall performance.

> ... one can perform a heuristic search in the space of the possible different ensemble subsets while evaluating the collective merit of a candidate subset.
>
> — Page 123, *Pattern Classification Using Ensemble Methods*, 2010.

Now that we are familiar with ensemble selection methods, let's explore how we might implement ensemble pruning and ensemble growing in scikit-learn.

27.3 Baseline Models and Voting

Before we dive into developing growing and pruning ensembles, let's first establish a dataset and baseline. We will use a synthetic binary classification problem as the basis for this investigation, defined by the `make_classification()` function with 5,000 examples and 20 numerical input features. The example below defines the dataset and summarizes its size.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=5000, n_features=20, n_informative=10, n_redundant=10,
    random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 27.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset in a repeatable manner and reports the number of rows and input features, matching our expectations.

```
(5000, 20) (5000,)
```

Listing 27.2: Example output from creating the synthetic classification dataset.

Next, we can choose some candidate models that will provide the basis for our ensemble. We will use five standard machine learning models, including logistic regression, naive Bayes, decision tree, support vector machine, and a k-nearest neighbor algorithm. First, we can define a function that will create each model with default hyperparameters. Each model will be defined as a tuple with a name and the model object, then added to a list. This is a helpful structure both for enumerating the models with their names for standalone evaluation and for later use in an ensemble. The `get_models()` function below implements this and returns the list of models to consider.

```
# get a list of models to evaluate
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
```

```
  models.append(('tree', DecisionTreeClassifier()))
  models.append(('nb', GaussianNB()))
  models.append(('svm', SVC(probability=True)))
  return models
```

Listing 27.3: Example of a function for defining standalone classification models.

We can then define a function that takes a single model and the dataset and evaluates the performance of the model on the dataset. We will evaluate a model using repeated stratified k-fold cross-validation with 10 folds and three repeats, a good practice in machine learning. The `evaluate_model()` function below implements this and returns a list of scores across all folds and repeats.

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

Listing 27.4: Example of a function for evaluating a model on a given dataset.

We can then create the list of models and enumerate them, reporting the performance of each on the synthetic dataset in turn. Tying this together, the complete example is listed below.

```
# evaluate standard models on the synthetic dataset
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.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=5000, n_features=20, n_informative=10,
      n_redundant=10, random_state=1)
  return X, y

# get a list of models to evaluate
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('tree', DecisionTreeClassifier()))
  models.append(('nb', GaussianNB()))
  models.append(('svm', SVC(probability=True)))
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
```

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

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models:
  # evaluate model
  scores = evaluate_model(model, X, y)
  # store results
  results.append(scores)
  names.append(name)
  # summarize result
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 27.5: Example of evaluating standalone models on the synthetic classification dataset.

Running the example evaluates each standalone machine learning algorithm on the synthetic binary classification dataset.

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

In this case, we can see that both the KNN and SVM models perform the best on this dataset, achieving a mean classification accuracy of about 95.3 percent. These results provide a baseline in performance that we require any ensemble to exceed in order to be considered useful on this dataset.

```
>lr 0.856 (0.014)
>knn 0.953 (0.008)
>tree 0.867 (0.014)
>nb 0.847 (0.021)
>svm 0.953 (0.010)
```

Listing 27.6: Example output from evaluating standalone models on the synthetic classification dataset.

A figure is created showing box and whisker plots of the distribution of accuracy scores for each algorithm. We can see that the KNN and SVM algorithms perform much better than the other algorithms, although all algorithms are skillful in different ways. This may make them good candidates to consider in an ensemble.

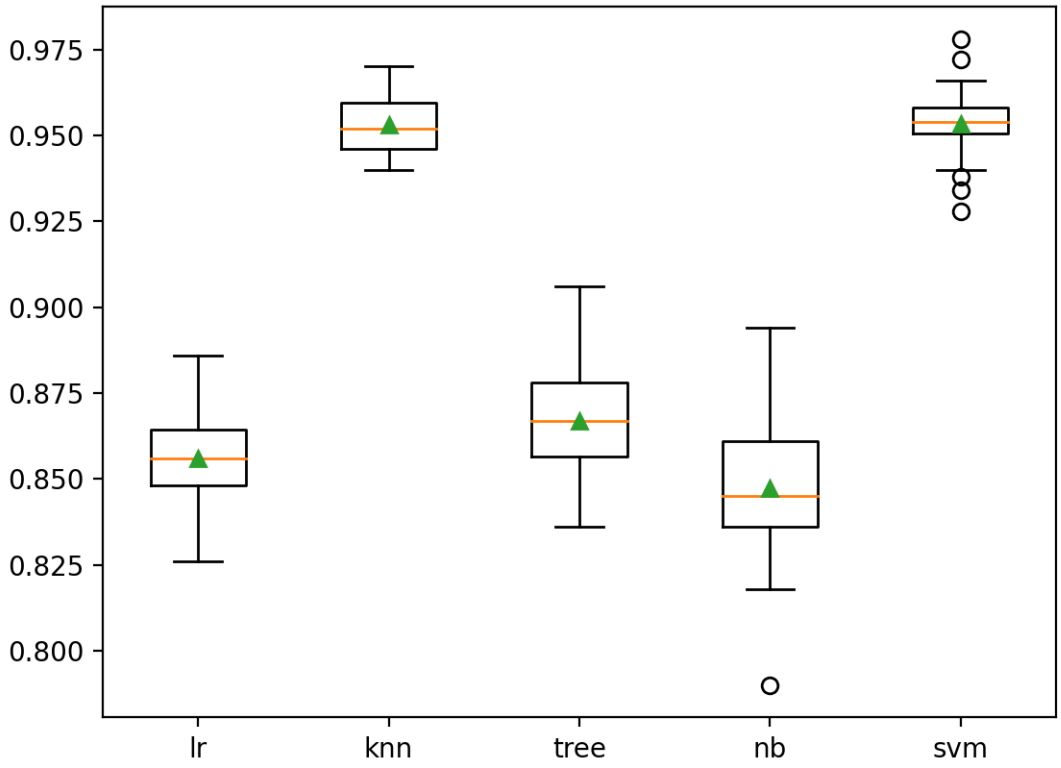

Figure 27.1: Box and Whisker Plots of Classification Accuracy for Standalone Machine Learning Models.

Next, we need to establish a baseline ensemble that uses all models. This will provide a point of comparison with growing and pruning methods that seek better performance with a smaller subset of models. In this case, we will use a voting ensemble with soft voting (introduced in Chapter 25). This means that each model will predict probabilities and the probabilities will be summed by the ensemble model to choose a final output prediction for each input sample. This can be achieved using the `VotingClassifier` class where the members are set via the `estimators` argument, which expects a list of models where each model is a tuple with a name and configured model object, just as we defined in the previous section. We can then set the type of voting to perform via the `voting` argument, which in this case is set to `'soft'`.

```
...
# create the ensemble
ensemble = VotingClassifier(estimators=models, voting='soft')
```

Listing 27.7: Example of a defining a voting ensemble with soft voting.

Tying this together, the example below evaluates a voting ensemble of all five models on the synthetic binary classification dataset.

```
# example of a voting ensemble with soft voting of ensemble members
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.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=5000, n_features=20, n_informative=10,
      n_redundant=10, random_state=1)
  return X, y

# get a list of models to evaluate
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('tree', DecisionTreeClassifier()))
  models.append(('nb', GaussianNB()))
  models.append(('svm', SVC(probability=True)))
  return models

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# create the ensemble
ensemble = VotingClassifier(estimators=models, voting='soft')
# define the evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate the ensemble
scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize the result
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 27.8: Example of evaluating a voting ensemble on the synthetic classification dataset.

Running the example evaluates the soft voting ensemble of all models using repeated stratified k-fold cross-validation and reports the mean accuracy across all folds and repeats.

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

In this case, we can see that the voting ensemble achieved a mean accuracy of about 92.8 percent. This is lower than SVM and KNN models used alone that achieved an accuracy of about 95.3 percent. This result highlights that a simple voting ensemble of all models results in a model with higher complexity and worse performance in this case. Perhaps we can find a subset of members that performs better than any single model and has lower complexity than simply using all models.

```
Mean Accuracy: 0.928 (0.012)
```

Listing 27.9: Example output from evaluating a voting ensemble on the synthetic classification dataset.

Next, we will explore pruning members from the ensemble.

27.4 Ensemble Pruning Example

In this section, we will explore how to develop a greedy ensemble pruning algorithm from scratch. We will use a greedy algorithm in this case, which is straightforward to implement. This involves removing one member from the ensemble and evaluating the performance and repeating this for each member in the ensemble. The member that, if removed, results in the best improvement in performance is then permanently removed from the ensemble and the process repeats. This continues until no further improvements can be achieved.

It is referred to as a *greedy* algorithm because it seeks the best improvement at each step. It is possible that the best combination of members is not on the path of greedy improvements, in which case the greedy algorithm will not find it and a global optimization algorithm could be used instead. First, we can define a function to evaluate a candidate list of models. This function will take the list of models and the dataset and construct a voting ensemble from the list of models and evaluate its performance using repeated stratified k-fold cross-validation, returning the mean classification accuracy. This function can be used to evaluate each candidate's removal from the ensemble. The `evaluate_ensemble()` function below implements this.

```
# evaluate a list of models
def evaluate_ensemble(models, X, y):
  # check for no models
  if len(models) == 0:
    return 0.0
  # create the ensemble
  ensemble = VotingClassifier(estimators=models, voting='soft')
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the ensemble
  scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # return mean score
  return mean(scores)
```

Listing 27.10: Example of a function for evaluating an ensemble of a given list of models.

Next, we can define a function that performs a single round of pruning. First, a baseline in performance is established with all models that are currently in the ensemble. Then the list of models is enumerated and each is removed in turn, and the effect of removing the model is evaluated on the ensemble. If the removal results in an improvement in performance, the new score and specific model that was removed is recorded. Importantly, the trial removal is performed on a copy of the list of models, not on the main list of models itself. This is to ensure we only remove an ensemble member from the list once we know it will result in the best possible improvement from all the members that could potentially be removed at the current step.

The prune_round() function below implements this given the list of current models in the ensemble and dataset, and returns the improvement in score (if any) and the best model to remove to achieve that score.

```
# perform a single round of pruning the ensemble
def prune_round(models_in, X, y):
  # establish a baseline
  baseline = evaluate_ensemble(models_in, X, y)
  best_score, removed = baseline, None
  # enumerate removing each candidate and see if we can improve performance
  for m in models_in:
    # copy the list of chosen models
    dup = models_in.copy()
    # remove this model
    dup.remove(m)
    # evaluate new ensemble
    result = evaluate_ensemble(dup, X, y)
    # check for new best
    if result > best_score:
      # store the new best
      best_score, removed = result, m
  return best_score, removed
```

Listing 27.11: Example of a function for the greedy pruning of ensemble members.

Next, we need to drive the pruning process. This involves running a round of pruning until no further improvement in accuracy is achieved by calling the prune_round() function repeatedly. If the function returns None for the model to be removed, we know that no single greedy improvement is possible and we can return the final list of models. Otherwise, the chosen model is removed from the ensemble and the process continues. The prune_ensemble() function below implements this and returns the models to use in the final ensemble and the score that it achieved via our evaluation procedure.

```
# prune an ensemble from scratch
def prune_ensemble(models, X, y):
  best_score = 0.0
  # prune ensemble until no further improvement
  while True:
    # remove one model to the ensemble
    score, removed = prune_round(models, X, y)
    # check for no improvement
    if removed is None:
      print('>no further improvement')
      break
    # keep track of best score
    best_score = score
    # remove model from the list
    models.remove(removed)
    # report results along the way
    print('>%.3f (removed: %s)' % (score, removed[0]))
  return best_score, models
```

Listing 27.12: Example of a function for driving the ensemble pruning process.

We can tie all of this together into an example of ensemble pruning on the synthetic binary classification dataset.

```
# example of ensemble pruning for classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=5000, n_features=20, n_informative=10,
      n_redundant=10, random_state=1)
  return X, y

# get a list of models to evaluate
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('tree', DecisionTreeClassifier()))
  models.append(('nb', GaussianNB()))
  models.append(('svm', SVC(probability=True)))
  return models

# evaluate a list of models
def evaluate_ensemble(models, X, y):
  # check for no models
  if len(models) == 0:
    return 0.0
  # create the ensemble
  ensemble = VotingClassifier(estimators=models, voting='soft')
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the ensemble
  scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # return mean score
  return mean(scores)

# perform a single round of pruning the ensemble
def prune_round(models_in, X, y):
  # establish a baseline
  baseline = evaluate_ensemble(models_in, X, y)
  best_score, removed = baseline, None
  # enumerate removing each candidate and see if we can improve performance
  for m in models_in:
    # copy the list of chosen models
    dup = models_in.copy()
    # remove this model
    dup.remove(m)
    # evaluate new ensemble
    result = evaluate_ensemble(dup, X, y)
```

```
    # check for new best
    if result > best_score:
      # store the new best
      best_score, removed = result, m
  return best_score, removed

# prune an ensemble from scratch
def prune_ensemble(models, X, y):
  best_score = 0.0
  # prune ensemble until no further improvement
  while True:
    # remove one model to the ensemble
    score, removed = prune_round(models, X, y)
    # check for no improvement
    if removed is None:
      print('>no further improvement')
      break
    # keep track of best score
    best_score = score
    # remove model from the list
    models.remove(removed)
    # report results along the way
    print('>%.3f (removed: %s)' % (score, removed[0]))
  return best_score, models

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# prune the ensemble
score, model_list = prune_ensemble(models, X, y)
names = ','.join([n for n,_ in model_list])
print('Models: %s' % names)
print('Final Mean Accuracy: %.3f' % score)
```

Listing 27.13: Example of evaluating ensemble pruning on the synthetic classification dataset.

Running the example performs the ensemble pruning process, reporting which model was removed each round and the accuracy of the model once the model was removed.

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

In this case, we can see that three rounds of pruning were performed, removing the naive Bayes, decision tree, and logistic regression algorithms, leaving only the SVM and KNN algorithms that achieved a mean classification accuracy of about 95.7 percent. This is better than the 95.3 percent achieved by SVM and KNN used in a standalone manner, and clearly better than combining all models together. The final list of models could then be used in a new final voting ensemble model via the `estimators` argument, fit on the entire dataset and used to make a prediction on new data.

```
>0.939 (removed: nb)
>0.948 (removed: tree)
```

```
>0.957 (removed: lr)
>no further improvement
Models: knn,svm
Final Mean Accuracy: 0.957
```

Listing 27.14: Example output from evaluating ensemble pruning on the synthetic classification dataset.

Now that we are familiar with developing and evaluating an ensemble pruning method, let's look at the reverse case of growing the ensemble members.

27.5 Ensemble Growing Example

In this section, we will explore how to develop a greedy ensemble growing algorithm from scratch. The structure of greedily growing an ensemble is much like the greedy pruning of members, although in reverse. We start with an ensemble with no models and add a single model that has the best performance. Models are then added one by one only if they result in a lift in performance over the ensemble before the model was added. Much of the code is the same as the pruning case so we can focus on the differences.

First, we must define a function to perform one round of growing the ensemble. This involves enumerating all candidate models that could be added and evaluating the effect of adding each in turn to the ensemble. The single model that results in the biggest improvement is then returned by the function, along with its score. The `grow_round()` function below implements this behavior.

```
# perform a single round of growing the ensemble
def grow_round(models_in, models_candidate, X, y):
  # establish a baseline
  baseline = evaluate_ensemble(models_in, X, y)
  best_score, addition = baseline, None
  # enumerate adding each candidate and see if we can improve performance
  for m in models_candidate:
    # copy the list of chosen models
    dup = models_in.copy()
    # add the candidate
    dup.append(m)
    # evaluate new ensemble
    result = evaluate_ensemble(dup, X, y)
    # check for new best
    if result > best_score:
      # store the new best
      best_score, addition = result, m
  return best_score, addition
```

Listing 27.15: Example of a function for the greedy growing of ensemble members.

Next, we need a function to drive the growing procedure. This involves a loop that runs rounds of growing until no further additions can be made resulting in an improvement in model performance. For each addition that can be made, the main list of models in the ensemble is updated and the list of models currently in the ensemble is reported along with the performance. The `grow_ensemble()` function implements this and returns the list of models greedily determined to result in the best performance along with the expected mean accuracy.

```
# grow an ensemble from scratch
def grow_ensemble(models, X, y):
  best_score, best_list = 0.0, list()
  # grow ensemble until no further improvement
  while True:
    # add one model to the ensemble
    score, addition = grow_round(best_list, models, X, y)
    # check for no improvement
    if addition is None:
      print('>no further improvement')
      break
    # keep track of best score
    best_score = score
    # remove new model from the list of candidates
    models.remove(addition)
    # add new model to the list of models in the ensemble
    best_list.append(addition)
    # report results along the way
    names = ','.join([n for n,_ in best_list])
    print('>%.3f (%s)' % (score, names))
  return best_score, best_list
```

Listing 27.16: Example of a function for driving the ensemble growing process.

Tying this together, the complete example of greedy ensemble growing on the synthetic binary classification dataset is listed below.

```
# example of ensemble growing for classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=5000, n_features=20, n_informative=10,
      n_redundant=10, random_state=1)
  return X, y

# get a list of models to evaluate
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('tree', DecisionTreeClassifier()))
  models.append(('nb', GaussianNB()))
  models.append(('svm', SVC(probability=True)))
  return models

# evaluate a list of models
```

```
def evaluate_ensemble(models, X, y):
  # check for no models
  if len(models) == 0:
    return 0.0
  # create the ensemble
  ensemble = VotingClassifier(estimators=models, voting='soft')
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the ensemble
  scores = cross_val_score(ensemble, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  # return mean score
  return mean(scores)

# perform a single round of growing the ensemble
def grow_round(models_in, models_candidate, X, y):
  # establish a baseline
  baseline = evaluate_ensemble(models_in, X, y)
  best_score, addition = baseline, None
  # enumerate adding each candidate and see if we can improve performance
  for m in models_candidate:
    # copy the list of chosen models
    dup = models_in.copy()
    # add the candidate
    dup.append(m)
    # evaluate new ensemble
    result = evaluate_ensemble(dup, X, y)
    # check for new best
    if result > best_score:
      # store the new best
      best_score, addition = result, m
  return best_score, addition

# grow an ensemble from scratch
def grow_ensemble(models, X, y):
  best_score, best_list = 0.0, list()
  # grow ensemble until no further improvement
  while True:
    # add one model to the ensemble
    score, addition = grow_round(best_list, models, X, y)
    # check for no improvement
    if addition is None:
      print('>no further improvement')
      break
    # keep track of best score
    best_score = score
    # remove new model from the list of candidates
    models.remove(addition)
    # add new model to the list of models in the ensemble
    best_list.append(addition)
    # report results along the way
    names = ','.join([n for n,_ in best_list])
    print('>%.3f (%s)' % (score, names))
  return best_score, best_list

# define dataset
X, y = get_dataset()
```

```
# get the models to evaluate
models = get_models()
# grow the ensemble
score, model_list = grow_ensemble(models, X, y)
names = ','.join([n for n,_ in model_list])
print('Models: %s' % names)
print('Final Mean Accuracy: %.3f' % score)
```

Listing 27.17: Example of evaluating ensemble growing on the synthetic classification dataset.

Running the example incrementally adds one model at a time to the ensemble and reports the mean classification accuracy of the ensemble of the models.

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

In this case, we can see that ensemble growing found the same solution as greedy ensemble pruning where an ensemble of SVM and KNN achieved a mean classification accuracy of about 95.6 percent, an improvement over any single standalone model and over combining all models.

```
>0.953 (svm)
>0.956 (svm,knn)
>no further improvement
Models: svm,knn
Final Mean Accuracy: 0.956
```

Listing 27.18: Example output from evaluating ensemble growing on the synthetic classification dataset.

27.6 Further Reading

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

Books

- *Ensemble Methods: Foundations and Algorithms*, 2012.
  https://amzn.to/2TavTcy
- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7

APIs

- `sklearn.ensemble.VotingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

27.7 Summary

In this tutorial, you discovered how to develop ensemble selection algorithms from scratch. Specifically, you learned:

- Ensemble selection involves choosing a subset of ensemble members that results in lower complexity than using all members and sometimes better performance.
- How to develop and evaluate a greedy ensemble pruning algorithm for classification.
- How to develop and evaluate an algorithm for greedily growing an ensemble from scratch.

Next

In the next section, we will take a closer look at the stacking ensemble algorithm.

Chapter 28

Stacking Ensemble

Stacking or Stacked Generalization is an ensemble machine learning algorithm. It uses a meta-learning algorithm to learn how to best combine the predictions from two or more base machine learning algorithms. The benefit of stacking is that it can harness the capabilities of a range of well-performing models on a classification or regression task and make predictions that have better performance than any single model in the ensemble. In this tutorial, you will discover the stacked generalization ensemble or stacking in Python. After completing this tutorial, you will know:

- Stacking is an ensemble machine learning algorithm that learns how to best combine the predictions from multiple well-performing machine learning models.
- The scikit-learn library provides a standard implementation of the stacking ensemble in Python.
- How to use stacking ensembles for regression and classification predictive modeling.

Let's get started.

28.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Stacked Generalization
2. Develop Stacking Ensembles
3. Stacking for Classification
4. Stacking for Regression
5. Common Questions

28.2 Stacked Generalization

Stacked Generalization or *stacking* for short is an ensemble machine learning algorithm. It involves combining the predictions from multiple machine learning models on the same dataset, like bagging and boosting. Stacking addresses the question:

- Given multiple machine learning models that are skillful on a problem, but in different ways, how do you choose which model to use (trust)?

The approach to this question is to use another machine learning model that learns when to use or trust each model in the ensemble.

> Stacking is a general procedure where a learner is trained to combine the individual learners. Here, the individual learners are called the first-level learners, while the combiner is called the second-level learner, or meta-learner.
>
> — Page 83, *Ensemble Methods*, 2012.

The architecture of a stacking model involves two or more base models, often referred to as level-0 models, and a meta-model that combines the predictions of the base models, referred to as a level-1 model.

- **Level-0 Models (*Base-Models*)**: Models fit on the training data and whose predictions are compiled.
- **Level-1 Model (*Meta-Model*)**: Model that learns how to best combine the predictions of the base models.

The meta-model is trained on the predictions made by base models on out-of-sample data. That is, data not used to train the base models is fed to the base models, predictions are made, and these predictions, along with the expected outputs, provide the input and output pairs of the training dataset used to fit the meta-model.

> ... we reserve some instances to form the training data for the level-1 learner and build level-0 classifiers from the remaining data. Once the level-0 classifiers have been built they are used to classify the instances in the holdout set, forming the level-1 training data.
>
> — Page 498, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

The outputs from the base models used as input to the meta-model may be real values in the case of regression, and probability values, probability like values, or class labels in the case of classification.

> ... most learning schemes are able to output probabilities for every class label instead of making a single categorical prediction. This can be exploited to improve the performance of stacking by using the probabilities to form the level-1 data.
>
> — Page 498, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

The most common approach to preparing the training dataset for the meta-model is via k-fold cross-validation of the base models, where the out-of-fold predictions are used as the basis for the training dataset for the meta-model. The training data for the meta-model may also include the inputs to the base models, e.g. input elements of the training data. This can provide an additional context to the meta-model as to how to best combine the predictions from the meta-model.

Once the training dataset is prepared for the meta-model, the meta-model can be trained in isolation on this dataset, and the base models can be trained on the entire original training dataset. Stacking is appropriate when multiple different machine learning models have skill on a dataset, but have skill in different ways. Another way to say this is that the predictions made by the models or the errors in predictions made by the models are uncorrelated or have a low correlation.

> The first-level learners are often generated by applying different learning algorithms, and so, stacked ensembles are often heterogeneous
>
> — Page 83, *Ensemble Methods*, 2012.

Base-models are often complex and diverse. As such, it is often a good idea to use a range of models that make very different assumptions about how to solve the predictive modeling task, such as linear models, decision trees, support vector machines, neural networks, and more. Other ensemble algorithms may also be used as base models, such as random forests. The meta-model is often simple, providing a smooth interpretation of the predictions made by the base models. As such, linear models are often used as the meta-model, such as linear regression for regression tasks (predicting a numeric value) and logistic regression for classification tasks (predicting a class label). Although this is common, it is not required.

> ... because most of the work is already done by the level-0 learners, the level-1 classifier is basically just an arbiter and it makes sense to choose a rather simple algorithm for this purpose. [...] Simple linear models or trees with linear models at the leaves usually work well.
>
> — Page 499, *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.

The use of a simple linear model as the meta-model often gives stacking the colloquial name **blending**. As in the prediction is a weighted average or blending of the predictions made by the base models. Alternately, blending may also refer to using base-model predictions on a hold out dataset instead of hold out cross-validation folds to train the meta-model. We will explore blending in depth in Chapter 29. The **super learner ensemble** may be considered a specialized type of stacking that explicitly uses k-fold cross-validation of base models to train the meta-model. We will explore the super learner ensemble in depth in Chapter 30.

Stacking is designed to improve modeling performance, although is not guaranteed to result in an improvement in all cases. Achieving an improvement in performance depends on the complexity of the problem and whether it is sufficiently well represented by the training data and complex enough that there is more to learn by combining predictions. It is also dependent upon the choice of base models and whether they are sufficiently skillful and sufficiently uncorrelated in their predictions (or errors). If a base-model performs as well as or better than the stacking ensemble, the base-model should be used instead, given its lower complexity (e.g. it's simpler to describe, train and maintain).

28.3 Develop Stacking Ensembles

The scikit-learn Python machine learning library provides an implementation of stacking ensembles for machine learning via the `StackingRegressor` and `StackingClassifier` classes. Both models operate the same way and take the same arguments. Using the model requires that you specify a list of estimators (level-0 models), and a final estimator (level-1 or meta-model). A list of level-0 models or base models is provided via the *estimators* argument. This is a Python list where each element in the list is a tuple with the name of the model and the configured model instance. For example, below defines two level-0 models:

```
...
models = [('lr',LogisticRegression()),('svm',SVC())
stacking = StackingClassifier(estimators=models)
```

Listing 28.1: Example of defining a stacking ensemble with a default meta-model.

Each model in the list may also be a `Pipeline`, including any data preparation required by the model prior to fitting the model on the training dataset. For example:

```
...
models = [('lr',LogisticRegression()),('svm',make_pipeline(StandardScaler(),SVC()))
stacking = StackingClassifier(estimators=models)
```

Listing 28.2: Example of defining a stacking ensemble where one member is a pipeline.

The level-1 model or meta-model is provided via the `final_estimator` argument. By default, this is set to `LinearRegression` for regression and `LogisticRegression` for classification, and these are sensible defaults that you probably do not want to change. The dataset for the meta-model is prepared using cross-validation. By default, 5-fold cross-validation is used, although this can be changed via the `cv` argument and set to either a number (e.g. 10 for 10-fold cross-validation) or a cross-validation object (e.g. `StratifiedKFold`). Sometimes, better performance can be achieved if the dataset prepared for the meta-model also includes inputs to the level-0 models, e.g. the input training data. This can be achieved by setting the `passthrough` argument to `True` and is not enabled by default. Now that we are familiar with the stacking API in scikit-learn, let's look at some worked examples.

28.4 Stacking for Classification

In this section, we will look at using stacking for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 28.3: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 28.4: Example output from creating the synthetic classification dataset.

Next, we can evaluate a suite of different machine learning models on the dataset. Specifically, we will evaluate the following five algorithms:

- Logistic Regression.
- k-Nearest Neighbors.
- Decision Tree.
- Support Vector Machine.
- Naive Bayes.

Each algorithm will be evaluated using default model hyperparameters. The function `get_models()` below creates the models we wish to evaluate.

```
# get a list of models to evaluate
def get_models():
  models = dict()
  models['lr'] = LogisticRegression()
  models['knn'] = KNeighborsClassifier()
  models['cart'] = DecisionTreeClassifier()
  models['svm'] = SVC()
  models['bayes'] = GaussianNB()
  return models
```

Listing 28.5: Example of a function for defining standalone classification models.

Each model will be evaluated using repeated k-fold cross-validation. The `evaluate_model()` function below takes a model instance and returns a list of scores from three repeats of stratified 10-fold cross-validation.

```
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores
```

Listing 28.6: Example of a function for evaluating a provided classification model.

We can then report the mean performance of each algorithm and also create a box and whisker plot to compare the distribution of accuracy scores for each algorithm. Tying this together, the complete example is listed below.

```
# compare standalone models for binary classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
```

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=1)
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  models['lr'] = LogisticRegression()
  models['knn'] = KNeighborsClassifier()
  models['cart'] = DecisionTreeClassifier()
  models['svm'] = SVC()
  models['bayes'] = GaussianNB()
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 28.7: Example of evaluating standalone models on the synthetic classification dataset.

Running the example first reports the mean and standard deviation accuracy for each model.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation

procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

We can see that in this case, SVM performs the best with about 95.7 percent mean accuracy.

```
>lr 0.866 (0.029)
>knn 0.931 (0.025)
>cart 0.821 (0.050)
>svm 0.957 (0.020)
>bayes 0.833 (0.031)
```

Listing 28.8: Example output from evaluating standalone models on the synthetic classification dataset.

A box-and-whisker plot is then created comparing the distribution accuracy scores for each model, allowing us to clearly see that KNN and SVM perform better on average than LR, CART, and Bayes.

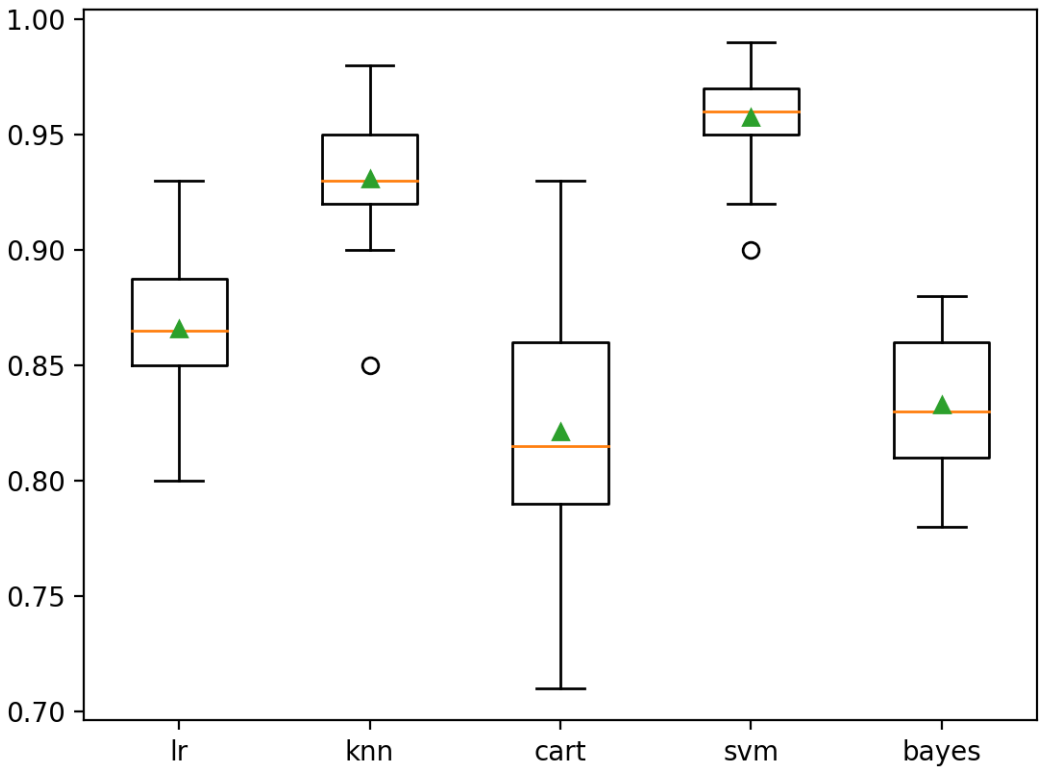

Figure 28.1: Box Plot of Standalone Model Accuracies for Binary Classification.

Here we have five different algorithms that perform well, presumably in different ways on this dataset. Next, we can try to combine these five models into a single ensemble model using stacking. We can use a logistic regression model to learn how to best combine the predictions from each of the separate five models. The `get_stacking()` function below defines

the `StackingClassifier` model by first defining a list of tuples for the five base models, then defining the logistic regression meta-model to combine the predictions from the base models using 5-fold cross-validation.

```
# get a stacking ensemble of models
def get_stacking():
  # define the base models
  level0 = list()
  level0.append(('lr', LogisticRegression()))
  level0.append(('knn', KNeighborsClassifier()))
  level0.append(('cart', DecisionTreeClassifier()))
  level0.append(('svm', SVC()))
  level0.append(('bayes', GaussianNB()))
  # define meta learner model
  level1 = LogisticRegression()
  # define the stacking ensemble
  model = StackingClassifier(estimators=level0, final_estimator=level1, cv=5)
  return model
```

Listing 28.9: Example of a function for defining a stacking ensemble for classification.

We can include the stacking ensemble in the list of models to evaluate, along with the standalone models.

```
# get a list of models to evaluate
def get_models():
  models = dict()
  models['lr'] = LogisticRegression()
  models['knn'] = KNeighborsClassifier()
  models['cart'] = DecisionTreeClassifier()
  models['svm'] = SVC()
  models['bayes'] = GaussianNB()
  models['stacking'] = get_stacking()
  return models
```

Listing 28.10: Example of a function for standalone and stacking ensemble classification models for comparison.

Our expectation is that the stacking ensemble will perform better than any single base-model. This is not always the case and if it is not the case, then the base-model should be used in favor of the ensemble model. The complete example of evaluating the stacking ensemble model alongside the standalone models is listed below.

```
# compare ensemble to each baseline classifier
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.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import StackingClassifier
from matplotlib import pyplot
```

```
# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
      n_redundant=5, random_state=1)
  return X, y

# get a stacking ensemble of models
def get_stacking():
  # define the base models
  level0 = list()
  level0.append(('lr', LogisticRegression()))
  level0.append(('knn', KNeighborsClassifier()))
  level0.append(('cart', DecisionTreeClassifier()))
  level0.append(('svm', SVC()))
  level0.append(('bayes', GaussianNB()))
  # define meta learner model
  level1 = LogisticRegression()
  # define the stacking ensemble
  model = StackingClassifier(estimators=level0, final_estimator=level1, cv=5)
  return model

# get a list of models to evaluate
def get_models():
  models = dict()
  models['lr'] = LogisticRegression()
  models['knn'] = KNeighborsClassifier()
  models['cart'] = DecisionTreeClassifier()
  models['svm'] = SVC()
  models['bayes'] = GaussianNB()
  models['stacking'] = get_stacking()
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
```

```
pyplot.show()
```

Listing 28.11: Example of comparing standalone models to a stacking ensemble on the synthetic classification dataset.

Running the example first reports the performance of each model. This includes the performance of each base-model, then the stacking ensemble.

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

In this case, we can see that the stacking ensemble appears to perform better than any single model on average, achieving an accuracy of about 96.4 percent.

```
>lr 0.866 (0.029)
>knn 0.931 (0.025)
>cart 0.820 (0.044)
>svm 0.957 (0.020)
>bayes 0.833 (0.031)
>stacking 0.964 (0.019)
```

Listing 28.12: Example output from comparing standalone models to a stacking ensemble on the synthetic classification dataset.

A box plot is created showing the distribution of model classification accuracies. Here, we can see that the mean and median accuracy for the stacking model sits slightly higher than the SVM model.

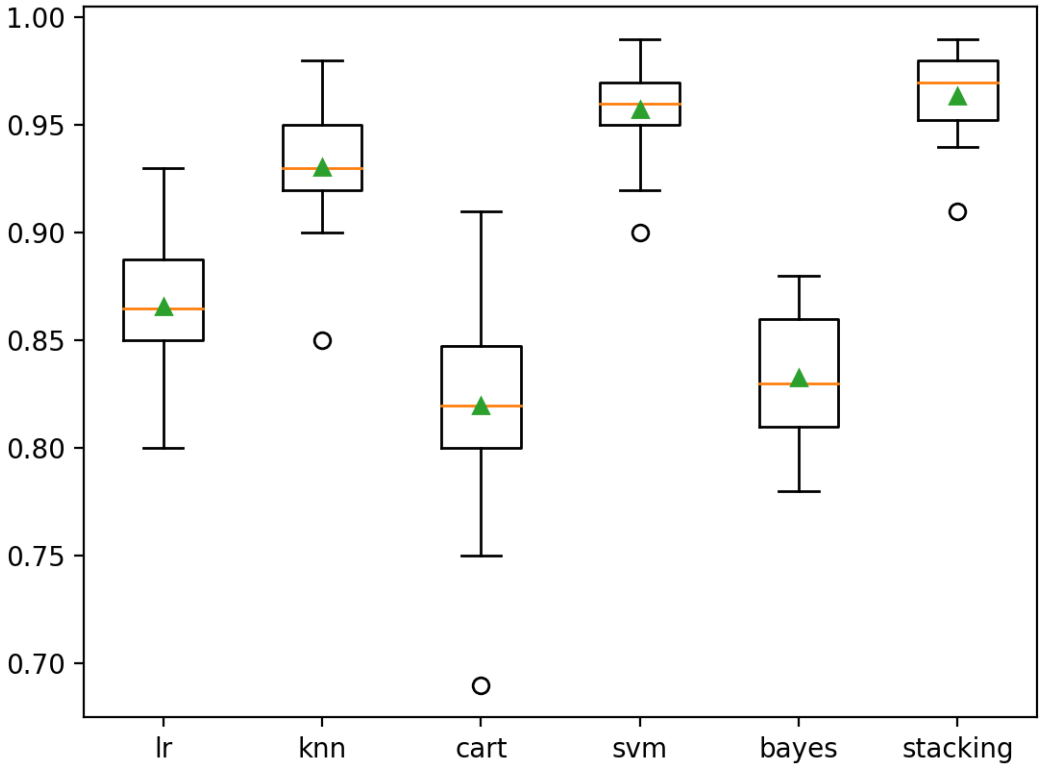

Figure 28.2: Box Plot of Standalone and Stacking Model Accuracies for Binary Classification.

If we choose a stacking ensemble as our final model, we can fit and use it to make predictions on new data just like any other model. First, the stacking ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our binary classification dataset.

```
# make a prediction with a stacking ensemble
from sklearn.datasets import make_classification
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=1)
# define the base models
level0 = list()
level0.append(('lr', LogisticRegression()))
level0.append(('knn', KNeighborsClassifier()))
level0.append(('cart', DecisionTreeClassifier()))
level0.append(('svm', SVC()))
level0.append(('bayes', GaussianNB()))
```

```
# define meta learner model
level1 = LogisticRegression()
# define the stacking ensemble
model = StackingClassifier(estimators=level0, final_estimator=level1, cv=5)
# fit the model on all available data
model.fit(X, y)
# make a prediction for one example
row = [2.47475454, 0.40165523, 1.68081787, 2.88940715, 0.91704519, -3.07950644, 4.39961206,
    0.72464273, -4.86563631, -6.06338084, -1.22209949, -0.4699618, 1.01222748, -0.6899355,
    -0.53000581, 6.86966784, -3.27211075, -6.59044146, -2.21290585, -3.139579]
yhat = model.predict([row])
# summarize prediction
print('Predicted Class: %d' % (yhat))
```

Listing 28.13: Example of making a prediction with a stacking ensemble for classification.

Running the example fits the stacking ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Class: 0
```

Listing 28.14: Example output making a prediction with a stacking ensemble for classification.

28.5 Stacking for Regression

In this section, we will look at using stacking for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 28.15: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 28.16: Example output from creating the synthetic classification dataset.

Next, we can evaluate a suite of different machine learning models on the dataset. Specifically, we will evaluate the following three algorithms:

- k-Nearest Neighbors.
- Decision Tree.
- Support Vector Regression.

The test dataset can be trivially solved using a linear regression model as the dataset was created using a linear model under the covers. As such, we will leave this model out of the example so we can demonstrate the benefit of the stacking ensemble method. Each algorithm will be evaluated using the default model hyperparameters. The function `get_models()` below creates the models we wish to evaluate.

```
# get a list of models to evaluate
def get_models():
  models = dict()
  models['knn'] = KNeighborsRegressor()
  models['cart'] = DecisionTreeRegressor()
  models['svm'] = SVR()
  return models
```

Listing 28.17: Example of a function for defining standalone regression models.

Each model will be evaluated using repeated k-fold cross-validation. The `evaluate_model()` function below takes a model instance and returns a list of scores from three repeats of 10-fold cross-validation.

```
# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
  return scores
```

Listing 28.18: Example of a function for evaluating a provided regression model.

We can then report the mean performance of each algorithm and also create a box and whisker plot to compare the distribution of accuracy scores for each algorithm. In this case, model performance will be reported using the mean absolute error (MAE). Tying this together, the complete example is listed below.

Note: The scikit-learn API flips the sign of the MAE to transform it from minimizing error to maximizing negative error. This means that large magnitude positive errors become large negative errors (e.g. 100 becomes -100) and a perfect model has no error with a value of 0.0. It also means that we can safely ignore the sign of the mean MAE scores.

```
# compare machine learning models for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
      random_state=1)
```

```
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  models['knn'] = KNeighborsRegressor()
  models['cart'] = DecisionTreeRegressor()
  models['svm'] = SVR()
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 28.19: Example of evaluating standalone models on the synthetic regression dataset.

Running the example first reports the mean and standard deviation MAE for each model.

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

We can see that in this case, KNN performs the best with a mean negative MAE of about 100.

```
>knn -101.019 (7.161)
>cart -148.100 (11.039)
>svm -162.419 (12.565)
```

Listing 28.20: Example output from evaluating standalone models on the synthetic regression dataset.

A box-and-whisker plot is then created comparing the distribution negative MAE scores for each model.

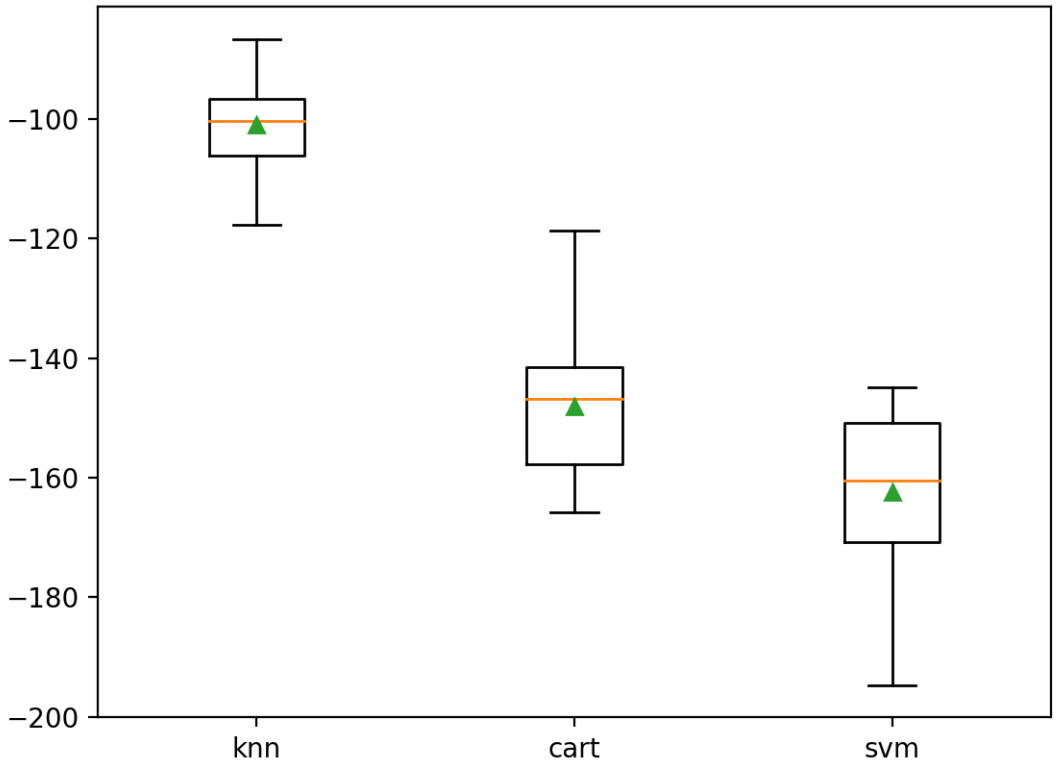

Figure 28.3: Box Plot of Standalone Model Negative Mean Absolute Error for Regression.

Here we have three different algorithms that perform well, presumably in different ways on this dataset. Next, we can try to combine these three models into a single ensemble model using stacking. We can use a linear regression model to learn how to best combine the predictions from each of the separate three models. The `get_stacking()` function below defines the `StackingRegressor` model by first defining a list of tuples for the three base models, then defining the linear regression meta-model to combine the predictions from the base models using 5-fold cross-validation.

```
# get a stacking ensemble of models
def get_stacking():
  # define the base models
  level0 = list()
  level0.append(('knn', KNeighborsRegressor()))
  level0.append(('cart', DecisionTreeRegressor()))
  level0.append(('svm', SVR()))
  # define meta learner model
  level1 = LinearRegression()
  # define the stacking ensemble
  model = StackingRegressor(estimators=level0, final_estimator=level1, cv=5)
  return model
```

Listing 28.21: Example of a function for defining a stacking ensemble for regression.

We can include the stacking ensemble in the list of models to evaluate, along with the standalone models.

```
# get a list of models to evaluate
def get_models():
  models = dict()
  models['knn'] = KNeighborsRegressor()
  models['cart'] = DecisionTreeRegressor()
  models['svm'] = SVR()
  models['stacking'] = get_stacking()
  return models
```

Listing 28.22: Example of a function for standalone and stacking ensemble regression models for comparison.

Our expectation is that the stacking ensemble will perform better than any single base-model. This is not always the case, and if it is not the case, then the base-model should be used in favor of the ensemble model. The complete example of evaluating the stacking ensemble model alongside the standalone models is listed below.

```
# compare ensemble to each standalone models for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import StackingRegressor
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
      random_state=1)
  return X, y

# get a stacking ensemble of models
def get_stacking():
  # define the base models
  level0 = list()
  level0.append(('knn', KNeighborsRegressor()))
  level0.append(('cart', DecisionTreeRegressor()))
  level0.append(('svm', SVR()))
  # define meta learner model
  level1 = LinearRegression()
  # define the stacking ensemble
  model = StackingRegressor(estimators=level0, final_estimator=level1, cv=5)
  return model

# get a list of models to evaluate
def get_models():
  models = dict()
  models['knn'] = KNeighborsRegressor()
```

```
  models['cart'] = DecisionTreeRegressor()
  models['svm'] = SVR()
  models['stacking'] = get_stacking()
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 28.23: Example of comparing standalone models to a stacking ensemble on the synthetic regression dataset.

Running the example first reports the performance of each model. This includes the performance of each base-model, then the stacking ensemble.

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

In this case, we can see that the stacking ensemble appears to perform better than any single model on average, achieving a mean negative MAE of about 56.

```
>knn -101.019 (7.161)
>cart -148.017 (10.635)
>svm -162.419 (12.565)
>stacking -56.893 (5.253)
```

Listing 28.24: Example output from comparing standalone models to a stacking ensemble on the synthetic regression dataset.

A box plot is created showing the distribution of model error scores. Here, we can see that the mean and median scores for the stacking model sit much higher than any individual model.

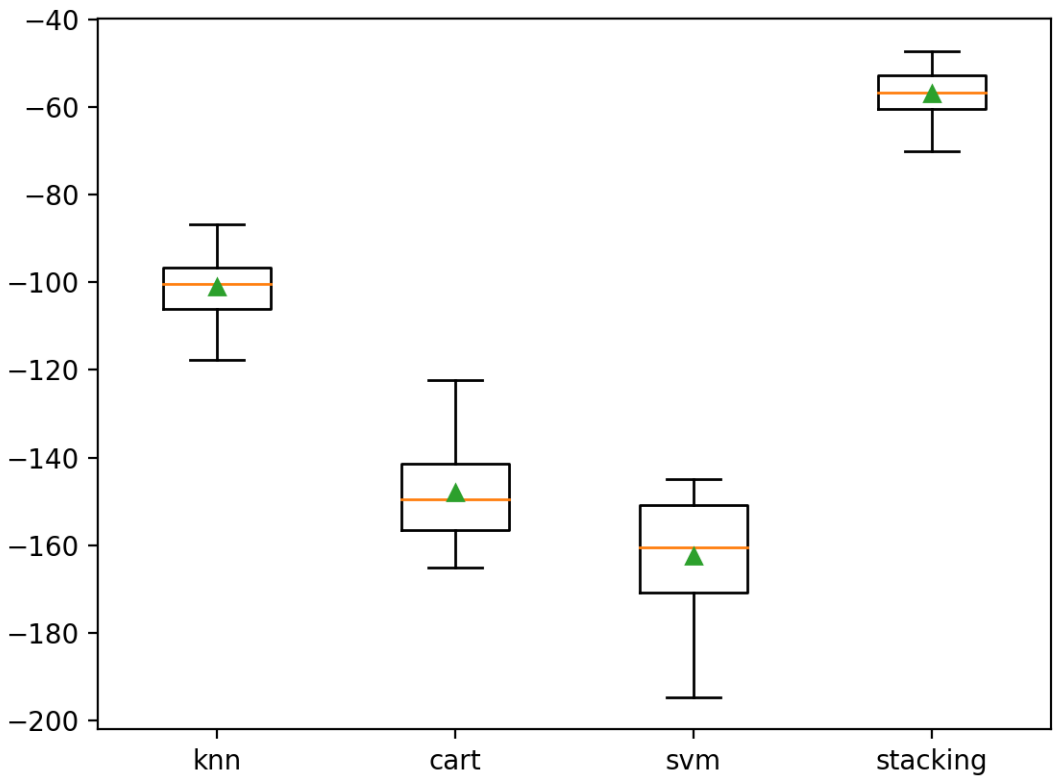

Figure 28.4: Box Plot of Standalone and Stacking Model Negative Mean Absolute Error for Regression.

If we choose a stacking ensemble as our final model, we can fit and use it to make predictions on new data just like any other model. First, the stacking ensemble is fit on all available data, then the `predict()` function can be called to make predictions on new data. The example below demonstrates this on our regression dataset.

```
# make a prediction with a stacking ensemble
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import StackingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1,
    random_state=1)
# define the base models
level0 = list()
level0.append(('knn', KNeighborsRegressor()))
level0.append(('cart', DecisionTreeRegressor()))
level0.append(('svm', SVR()))
# define meta learner model
level1 = LinearRegression()
```

```
# define the stacking ensemble
model = StackingRegressor(estimators=level0, final_estimator=level1, cv=5)
# fit the model on all available data
model.fit(X, y)
# make a prediction for one example
row = [0.59332206, -0.56637507, 1.34808718, -0.57054047, -0.72480487, 1.05648449,
    0.77744852, 0.07361796, 0.88398267, 2.02843157, 1.01902732, 0.11227799, 0.94218853,
    0.26741783, 0.91458143, -0.72759572, 1.08842814, -0.61450942, -0.69387293, 1.69169009]
yhat = model.predict([row])
# summarize prediction
print('Predicted Value: %.3f' % (yhat))
```

Listing 28.25: Example of making a prediction with a stacking ensemble for regression.

Running the example fits the stacking ensemble model on the entire dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Predicted Value: 556.264
```

Listing 28.26: Example output from making a prediction with a stacking ensemble for regression.

28.6 Common Questions

In this section we will take a closer look at some common sticking points you may have with the stacking ensemble procedure.

Q. What are the inputs and outputs for the meta-model?

The meta-model takes in predictions from base-models as input and predicts the target for the training dataset as output:

- **Input**: Predictions from base-models.
- **Output**: Prediction for training dataset.

For example, if we had 50 base-models, then one input sample would be a vector with 50 values, each value in the vector representing a prediction from one of the base-models for one sample of the training dataset. If we had 1,000 examples (rows) in the training dataset and 50 models, then the input data for the meta-model would be 1,000 rows and 50 columns.

Q. Won't the meta-model overfit the training data?

Probably not (see Chapter 6). This is the trick of the stacked generalization procedure. The input to the meta-model is the out-of-fold (out-of-sample) predictions. In aggregate, the out-of-fold predictions for a model represent the model's skill or capability in making predictions on data not seen during training.

By training a meta-model on out-of-sample predictions of other models, the meta-model learns how to both correct the out-of-sample predictions for each model and to best combine the out-of-sample predictions from multiple models; actually, it does both tasks at the same time. Importantly, to get an idea of the true capability of the meta-model, it must be evaluated on new out-of-sample data. That is, data not used to train the base models.

Q. Why do we fit each base-model on the entire training dataset?

Each base-model is fit on the entire training dataset so that the model can be used later to make predictions on new examples not seen during training. This step is strictly not required until predictions are needed by the stacking ensemble.

Q. How do we make a prediction?

To make a prediction on a new sample (row of data), first, the row of data is provided as input to each base-model to generate a prediction from each model. The predictions from the base-models are then concatenated into a vector and provided as input to the meta-model. The meta-model then makes a final prediction for the row of data. We can summarize this procedure as follows:

1. Take a sample not seen by the models during training.
2. For each base-model:
   (a) Make a prediction given the sample.
   (b) Store prediction.
3. Concatenate predictions from submodel into a single vector.
4. Provide vector as input to the meta-model to make a final prediction.

28.7 Further Reading

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

Papers

- *Stacked Generalization*, 1992.
  https://www.sciencedirect.com/science/article/abs/pii/S0893608005800231
- *Issues in Stacked Generalization*, 1999.
  https://arxiv.org/abs/1105.5466

Books

- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7
- *Ensemble Methods*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Machine Learning*, 2012.
  https://amzn.to/2C7syo5

- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2NJT1L8
- *The Elements of Statistical Learning*, 2017.
  https://amzn.to/38tJjVt
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2RzlcOK

APIs

- `sklearn.ensemble.StackingClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html
- `sklearn.ensemble.StackingRegressor` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingRegressor.html

Articles

- Ensemble learning, Wikipedia.
  https://en.wikipedia.org/wiki/Ensemble_learning

28.8 Summary

In this tutorial, you discovered the stacked generalization ensemble or stacking in Python. Specifically, you learned:

- Stacking is an ensemble machine learning algorithm that learns how to best combine the predictions from multiple well-performing machine learning models.
- The scikit-learn library provides a standard implementation of the stacking ensemble in Python.
- How to use stacking ensembles for regression and classification predictive modeling.

Next

In the next section, we will take a closer look at an extension to stacking called the blending ensemble.

Chapter 29

Blending Ensemble

Blending is an ensemble machine learning algorithm. It is a colloquial name for stacked generalization or stacking ensemble where instead of fitting the meta-model on out-of-fold predictions made by the base-model, it is fit on predictions made on a holdout dataset. Blending was used to describe stacking models that combined many hundreds of predictive models by competitors in the One Million Dollar Netflix machine learning competition, and as such, remains a popular technique and name for stacking in competitive machine learning circles, such as the Kaggle community. In this tutorial, you will discover how to develop and evaluate a blending ensemble in Python. After completing this tutorial, you will know:

- Blending ensembles are a type of stacking where the meta-model is fit using predictions on a holdout validation dataset instead of out-of-fold predictions.
- How to develop a blending ensemble, including functions for training the model and making predictions on new data.
- How to evaluate blending ensembles for classification and regression predictive modeling problems.

Let's get started.

29.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Blending Ensemble
2. Develop a Blending Ensemble
3. Blending Ensemble for Classification
4. Blending Ensemble for Regression

29.2 Blending Ensemble

Blending is an ensemble machine learning technique that uses a machine learning model to learn how to best combine the predictions from multiple contributing ensemble member models. As such, blending is the same as stacked generalization, known as stacking, broadly conceived (introduced in Chapter 28). Often, blending and stacking are used interchangeably in the same paper or model description.

> Many machine learning practitioners have had success using stacking and related techniques to boost prediction accuracy beyond the level obtained by any of the individual models. In some contexts, stacking is also referred to as blending, and we will use the terms interchangeably here.
>
> — *Feature-Weighted Linear Stacking*, 2009.

The architecture of a stacking model involves two or more base models, often referred to as level-0 models, and a meta-model that combines the predictions of the base models, referred to as a level-1 model. The meta-model is trained on the predictions made by base models on out-of-sample data.

- **Level-0 Models (*Base-Models*)**: Models fit on the training data and whose predictions are compiled.
- **Level-1 Model (*Meta-Model*)**: Model that learns how to best combine the predictions of the base models.

Nevertheless, blending has specific connotations for how to construct a stacking ensemble model. Blending may suggest developing a stacking ensemble where the base-models are machine learning models of any type, and the meta-model is a linear model that *blends* the predictions of the base-models. For example, a linear regression model when predicting a numerical value or a logistic regression model when predicting a class label would calculate a weighted sum of the predictions made by base models and would be considered a blending of predictions.

- **Blending Ensemble**: Use of a linear model, such as linear regression or logistic regression, as the meta-model in a stacking ensemble.

Blending was the term commonly used for stacking ensembles during the Netflix prize in 2009. The prize involved teams seeking movie recommendation predictions that performed better than the native Netflix algorithm and a US$1M prize was awarded to the team that achieved a 10 percent performance improvement.

> Our $RMSE = 0.8643^2$ solution is a linear blend of over 100 results. [...] Throughout the description of the methods, we highlight the specific predictors that participated in the final blended solution.
>
> — *The BellKor 2008 Solution to the Netflix Prize*, 2008.

As such, blending is a colloquial term for ensemble learning with a stacking-type architecture model. It is rarely, if ever, used in textbooks or academic papers, other than those related to competitive machine learning. Most commonly, blending is used to describe the specific application of stacking where the meta-model is trained on the predictions made by base-models on a hold-out validation dataset. In this context, stacking is reserved for a meta-model that is trained on out-of fold predictions during a cross-validation procedure.

- **Blending**: Stacking-type ensemble where the meta-model is trained on predictions made on a holdout dataset.
- **Stacking**: Stacking-type ensemble where the meta-model is trained on out-of-fold predictions made during k-fold cross-validation.

This distinction is common among the Kaggle competitive machine learning community.

> Blending is a word introduced by the Netflix winners. It is very close to stacked generalization, but a bit simpler and less risk of an information leak. [...] With blending, instead of creating out-of-fold predictions for the train set, you create a small holdout set of say 10% of the train set. The stacker model then trains on this holdout set only.
>
> — *Kaggle Ensemble Guide*, MLWave, 2015.

We will use this latter definition of blending. Next, let's look at how we can implement blending.

29.3 Develop a Blending Ensemble

The scikit-learn library does not natively support blending at the time of writing. Instead, we can implement it ourselves using scikit-learn models. First, we need to create a number of base models. These can be any models we like for a regression or classification problem. We can define a function `get_models()` that returns a list of models where each model is defined as a tuple with a name and the configured classifier or regression object. For example, for a classification problem, we might use a logistic regression, kNN, decision tree, SVM, and Naive Bayes model.

```
# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('svm', SVC(probability=True)))
  models.append(('bayes', GaussianNB()))
  return models
```

Listing 29.1: Example of a function for defining a list of classification models to include in the blending ensemble.

Next, we need to fit the blending model. Recall that the base models are fit on a training dataset. The meta-model is fit on the predictions made by each base-model on a holdout dataset. First, we can enumerate the list of models and fit each in turn on the training dataset. Also in this loop, we can use the fit model to make a prediction on the hold out (validation) dataset and store the predictions for later.

```
...
# fit all models on the training set and predict on hold out set
meta_X = list()
for _, model in models:
  # fit in training set
  model.fit(X_train, y_train)
  # predict on hold out set
  yhat = model.predict(X_val)
  # reshape predictions into a matrix with one column
  yhat = yhat.reshape(len(yhat), 1)
  # store predictions as input for blending
  meta_X.append(yhat)
```

Listing 29.2: Example of evaluating base models on a validation dataset.

We now have `meta_X` that represents the input data that can be used to train the meta-model. Each column or feature represents the output of one base-model. Each row represents the one sample from the holdout dataset. We can use the `hstack()` function to ensure this dataset is a 2D NumPy array as expected by a machine learning model.

```
...
# create 2d array from predictions, each set is an input feature
meta_X = hstack(meta_X)
```

Listing 29.3: Example of creating a dataset for training a meta-model.

We can now train our meta-model. This can be any machine learning model we like, such as logistic regression for classification.

```
...
# define blending model
blender = LogisticRegression()
# fit on predictions from base models
blender.fit(meta_X, y_val)
```

Listing 29.4: Example of defining and fitting a meta-model.

We can tie all of this together into a function named `fit_ensemble()` that trains the blending model using a training dataset and holdout validation dataset.

```
# fit the blending ensemble
def fit_ensemble(models, X_train, X_val, y_train, y_val):
  # fit all models on the training set and predict on hold out set
  meta_X = list()
  for _, model in models:
    # fit in training set
    model.fit(X_train, y_train)
    # predict on hold out set
    yhat = model.predict(X_val)
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
```

```
    # store predictions as input for blending
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # define blending model
  blender = LogisticRegression()
  # fit on predictions from base models
  blender.fit(meta_X, y_val)
  return blender
```

Listing 29.5: Example of a function for fitting a blending ensemble.

The next step is to use the blending ensemble to make predictions on new data. This is a two-step process. The first step is to use each base-model to make a prediction. These predictions are then gathered together and used as input to the blending model to make the final prediction. We can use the same looping structure as we did when training the model. That is, we can collect the predictions from each base-model into a training dataset, stack the predictions together, and call `predict()` on the blender model with this meta-level dataset. The `predict_ensemble()` function below implements this. Given the list of fit base models, the fit blender ensemble, and a dataset (such as a test dataset or new data), it will return a set of predictions for the dataset.

```
# make a prediction with the blending ensemble
def predict_ensemble(models, blender, X_test):
  # make predictions with base models
  meta_X = list()
  for _, model in models:
    # predict with base model
    yhat = model.predict(X_test)
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
    # store prediction
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # predict
  return blender.predict(meta_X)
```

Listing 29.6: Example of a function for making a prediction with the blending ensemble.

We now have all of the elements required to implement a blending ensemble for classification or regression predictive modeling problems

29.4 Blending Ensemble for Classification

In this section, we will look at using blending for a classification problem. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 10,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_informative=15, n_redundant=5,
    random_state=7)
```

```
# summarize the dataset
print(X.shape, y.shape)
```

Listing 29.7: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(10000, 20) (10000,)
```

Listing 29.8: Example output from creating the synthetic classification dataset.

Next, we need to split the dataset up, first into train and test sets, and then the training set into a subset used to train the base models and a subset used to train the meta-model. In this case, we will use a 50-50 split for the train and test sets, then use a 67-33 split for train and validation sets.

```
...
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.5,
    random_state=1)
# split training set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# summarize data split
print('Train: %s, Val: %s, Test: %s' % (X_train.shape, X_val.shape, X_test.shape))
```

Listing 29.9: Example of defining the train, validation and test dataset.

We can then use the `get_models()` function from the previous section to create the classification models used in the ensemble. The `fit_ensemble()` function can then be called to fit the blending ensemble on the train and validation datasets and the `predict_ensemble()` function can be used to make predictions on the holdout dataset.

```
...
# create the base models
models = get_models()
# train the blending ensemble
blender = fit_ensemble(models, X_train, X_val, y_train, y_val)
# make predictions on test set
yhat = predict_ensemble(models, blender, X_test)
```

Listing 29.10: Example of creating, fitting and making a prediction with the blending ensemble.

Finally, we can evaluate the performance of the blending model by reporting the classification accuracy on the test dataset.

```
...
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Blending Accuracy: %.3f' % score)
```

Listing 29.11: Example of evaluating the predictions made by the blending ensemble.

Tying this all together, the complete example of evaluating a blending ensemble on the synthetic binary classification problem is listed below.

```
# blending ensemble for classification using hard voting
from numpy import hstack
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=10000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('svm', SVC()))
  models.append(('bayes', GaussianNB()))
  return models

# fit the blending ensemble
def fit_ensemble(models, X_train, X_val, y_train, y_val):
  # fit all models on the training set and predict on hold out set
  meta_X = list()
  for _, model in models:
    # fit in training set
    model.fit(X_train, y_train)
    # predict on hold out set
    yhat = model.predict(X_val)
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
    # store predictions as input for blending
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # define blending model
  blender = LogisticRegression()
  # fit on predictions from base models
  blender.fit(meta_X, y_val)
  return blender

# make a prediction with the blending ensemble
def predict_ensemble(models, blender, X_test):
  # make predictions with base models
  meta_X = list()
  for _, model in models:
    # predict with base model
    yhat = model.predict(X_test)
```

```
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
    # store prediction
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # predict
  return blender.predict(meta_X)

# define dataset
X, y = get_dataset()
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.5,
    random_state=1)
# split training set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# summarize data split
print('Train: %s, Val: %s, Test: %s' % (X_train.shape, X_val.shape, X_test.shape))
# create the base models
models = get_models()
# train the blending ensemble
blender = fit_ensemble(models, X_train, X_val, y_train, y_val)
# make predictions on test set
yhat = predict_ensemble(models, blender, X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Blending Accuracy: %.3f' % (score*100))
```

Listing 29.12: Example of evaluating the blending ensemble on the synthetic classification dataset.

Running the example first reports the shape of the train, validation, and test datasets, then the accuracy of the ensemble on the test dataset.

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

In this case, we can see that the blending ensemble achieved a classification accuracy of about 97.900 percent.

```
Train: (3350, 20), Val: (1650, 20), Test: (5000, 20)
Blending Accuracy: 97.900
```

Listing 29.13: Example output from evaluating the blending ensemble on the synthetic classification dataset.

In the previous example, crisp class label predictions were combined using the blending model. This is a type of hard voting. An alternative is to have each model predict class probabilities and use the meta-model to blend the probabilities. This is a type of soft voting and can result in better performance in some cases. First, we must configure the models to return probabilities, such as the SVM model.

```
# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('svm', SVC(probability=True)))
  models.append(('bayes', GaussianNB()))
  return models
```

Listing 29.14: Example of a function for defining classification models for predicting probabilities.

Next, we must change the base models to predict probabilities instead of crisp class labels. This can be achieved by calling the `predict_proba()` function in the `fit_ensemble()` function when fitting the base models.

```
...
# fit all models on the training set and predict on hold out set
meta_X = list()
for _, model in models:
  # fit in training set
  model.fit(X_train, y_train)
  # predict on hold out set
  yhat = model.predict_proba(X_val)
  # store predictions as input for blending
  meta_X.append(yhat)
```

Listing 29.15: Example of preparing the blending dataset using predicted class probabilities.

This means that the meta dataset used to train the meta-model will have n columns per classifier, where n is the number of classes in the prediction problem, two in our case. We also need to change the predictions made by the base models when using the blending model to make predictions on new data.

```
...
# make predictions with base models
meta_X = list()
for _, model in models:
  # predict with base model
  yhat = model.predict_proba(X_test)
  # store prediction
  meta_X.append(yhat)
```

Listing 29.16: Example of updating the prediction process for the blending ensemble to use class probabilities.

Tying this together, the complete example of using blending on predicted class probabilities for the synthetic binary classification problem is listed below.

```
# blending ensemble for classification using soft voting
from numpy import hstack
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=10000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('svm', SVC(probability=True)))
  models.append(('bayes', GaussianNB()))
  return models

# fit the blending ensemble
def fit_ensemble(models, X_train, X_val, y_train, y_val):
  # fit all models on the training set and predict on hold out set
  meta_X = list()
  for _, model in models:
    # fit in training set
    model.fit(X_train, y_train)
    # predict on hold out set
    yhat = model.predict_proba(X_val)
    # store predictions as input for blending
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # define blending model
  blender = LogisticRegression()
  # fit on predictions from base models
  blender.fit(meta_X, y_val)
  return blender

# make a prediction with the blending ensemble
def predict_ensemble(models, blender, X_test):
  # make predictions with base models
  meta_X = list()
  for _, model in models:
    # predict with base model
    yhat = model.predict_proba(X_test)
    # store prediction
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # predict
  return blender.predict(meta_X)

# define dataset
X, y = get_dataset()
# split dataset into train and test sets
```

```
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.5,
    random_state=1)
# split training set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# summarize data split
print('Train: %s, Val: %s, Test: %s' % (X_train.shape, X_val.shape, X_test.shape))
# create the base models
models = get_models()
# train the blending ensemble
blender = fit_ensemble(models, X_train, X_val, y_train, y_val)
# make predictions on test set
yhat = predict_ensemble(models, blender, X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Blending Accuracy: %.3f' % (score*100))
```

Listing 29.17: Example of evaluating the blending ensemble with class probabilities as input on the synthetic classification dataset.

Running the example first reports the shape of the train, validation, and test datasets, then the accuracy of the ensemble on the test dataset.

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

In this case, we can see that blending the class probabilities resulted in a lift in classification accuracy to about 98.240 percent.

```
Train: (3350, 20), Val: (1650, 20), Test: (5000, 20)
Blending Accuracy: 98.240
```

Listing 29.18: Example output from evaluating the blending ensemble with class probabilities as input on the synthetic classification dataset.

A blending ensemble is only effective if it is able to out-perform any single contributing model. We can confirm this by evaluating each of the base models in isolation. Each base-model can be fit on the entire training dataset (unlike the blending ensemble) and evaluated on the test dataset (just like the blending ensemble). The example below demonstrates this, evaluating each base-model in isolation.

```
# evaluate base models on the entire training dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

# get the dataset
def get_dataset():
```

```
  X, y = make_classification(n_samples=10000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('svm', SVC(probability=True)))
  models.append(('bayes', GaussianNB()))
  return models

# define dataset
X, y = get_dataset()
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.5,
    random_state=1)
# summarize data split
print('Train: %s, Test: %s' % (X_train_full.shape, X_test.shape))
# create the base models
models = get_models()
# evaluate standalone model
for name, model in models:
  # fit the model on the training dataset
  model.fit(X_train_full, y_train_full)
  # make a prediction on the test dataset
  yhat = model.predict(X_test)
  # evaluate the predictions
  score = accuracy_score(y_test, yhat)
  # report the score
  print('>%s Accuracy: %.3f' % (name, score*100))
```

Listing 29.19: Example of evaluating standalone models on the synthetic classification dataset.

Running the example first reports the shape of the full train and test datasets, then the accuracy of each base-model on the test dataset.

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

In this case, we can see that all models perform worse than the blended ensemble.

Interestingly, we can see that the SVM comes very close to achieving an accuracy of 98.200 percent compared to 98.240 achieved with the blending ensemble.

```
Train: (5000, 20), Test: (5000, 20)
>lr Accuracy: 87.800
>knn Accuracy: 97.380
>cart Accuracy: 88.200
>svm Accuracy: 98.200
>bayes Accuracy: 87.300
```

Listing 29.20: Example output from evaluating standalone models on the synthetic classification dataset.

We may choose to use a blending ensemble as our final model. This involves fitting the ensemble on the entire training dataset and making predictions on new examples. Specifically, the entire training dataset is split onto train and validation sets to train the base and meta-models respectively, then the ensemble can be used to make a prediction. The complete example of making a prediction on new data with a blending ensemble for classification is listed below.

```
# example of making a prediction with a blending ensemble for classification
from numpy import hstack
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=10000, n_features=20, n_informative=15,
      n_redundant=5, random_state=7)
  return X, y

# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LogisticRegression()))
  models.append(('knn', KNeighborsClassifier()))
  models.append(('cart', DecisionTreeClassifier()))
  models.append(('svm', SVC(probability=True)))
  models.append(('bayes', GaussianNB()))
  return models

# fit the blending ensemble
def fit_ensemble(models, X_train, X_val, y_train, y_val):
  # fit all models on the training set and predict on hold out set
  meta_X = list()
  for _, model in models:
    # fit in training set
    model.fit(X_train, y_train)
    # predict on hold out set
    yhat = model.predict_proba(X_val)
    # store predictions as input for blending
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # define blending model
  blender = LogisticRegression()
  # fit on predictions from base models
  blender.fit(meta_X, y_val)
  return blender

# make a prediction with the blending ensemble
def predict_ensemble(models, blender, X_test):
  # make predictions with base models
  meta_X = list()
```

```
  for _, model in models:
    # predict with base model
    yhat = model.predict_proba(X_test)
    # store prediction
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # predict
  return blender.predict(meta_X)

# define dataset
X, y = get_dataset()
# split dataset set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.33, random_state=1)
# summarize data split
print('Train: %s, Val: %s' % (X_train.shape, X_val.shape))
# create the base models
models = get_models()
# train the blending ensemble
blender = fit_ensemble(models, X_train, X_val, y_train, y_val)
# make a prediction on a new row of data
row = [-0.30335011, 2.68066314, 2.07794281, 1.15253537, -2.0583897, -2.51936601,
    0.67513028, -3.20651939, -1.60345385, 3.68820714, 0.05370913, 1.35804433, 0.42011397,
    1.4732839, 2.89997622, 1.61119399, 7.72630965, -2.84089477, -1.83977415, 1.34381989]
yhat = predict_ensemble(models, blender, [row])
# summarize prediction
print('Predicted Class: %d' % (yhat))
```

Listing 29.21: Example of making a prediction with a blending ensemble for classification.

Running the example fits the blending ensemble model on the dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Train: (6700, 20), Val: (3300, 20)
Predicted Class: 1
```

Listing 29.22: Example output from making a prediction with a blending ensemble for classification.

Next, let's explore how we might evaluate a blending ensemble for regression.

29.5 Blending Ensemble for Regression

In this section, we will look at using stacking for a regression problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 10,000 examples and 20 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=10000, n_features=20, n_informative=10, noise=0.3,
    random_state=7)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 29.23: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(10000, 20) (10000,)
```

Listing 29.24: Example output from creating the synthetic classification dataset.

Next, we can define the list of regression models to use as base models. In this case, we will use linear regression, kNN, decision tree, and SVM models.

```
# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LinearRegression()))
  models.append(('knn', KNeighborsRegressor()))
  models.append(('cart', DecisionTreeRegressor()))
  models.append(('svm', SVR()))
  return models
```

Listing 29.25: Example of a function for defining a list of regression models to include in the blending ensemble.

The `fit_ensemble()` function used to train the blending ensemble is unchanged from classification, other than the model used for blending must be changed to a regression model. We will use the linear regression model in this case.

```
...
# define blending model
blender = LinearRegression()
```

Listing 29.26: Example of defining the blending ensemble meta-model for regression.

Given that it is a regression problem, we will evaluate the performance of the model using an error metric, in this case, the mean absolute error, or MAE for short.

```
...
# evaluate predictions
score = mean_absolute_error(y_test, yhat)
print('Blending MAE: %.3f' % score)
```

Listing 29.27: Example of evaluating predictions made by the blending ensemble for regression.

Tying this together, the complete example of a blending ensemble for the synthetic regression predictive modeling problem is listed below.

```
# evaluate blending ensemble for regression
from numpy import hstack
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR

# get the dataset
def get_dataset():
```

```
  X, y = make_regression(n_samples=10000, n_features=20, n_informative=10, noise=0.3,
      random_state=7)
  return X, y

# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LinearRegression()))
  models.append(('knn', KNeighborsRegressor()))
  models.append(('cart', DecisionTreeRegressor()))
  models.append(('svm', SVR()))
  return models

# fit the blending ensemble
def fit_ensemble(models, X_train, X_val, y_train, y_val):
  # fit all models on the training set and predict on hold out set
  meta_X = list()
  for _, model in models:
    # fit in training set
    model.fit(X_train, y_train)
    # predict on hold out set
    yhat = model.predict(X_val)
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
    # store predictions as input for blending
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # define blending model
  blender = LinearRegression()
  # fit on predictions from base models
  blender.fit(meta_X, y_val)
  return blender

# make a prediction with the blending ensemble
def predict_ensemble(models, blender, X_test):
  # make predictions with base models
  meta_X = list()
  for _, model in models:
    # predict with base model
    yhat = model.predict(X_test)
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
    # store prediction
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # predict
  return blender.predict(meta_X)

# define dataset
X, y = get_dataset()
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.5,
    random_state=1)
# split training set into train and validation sets
```

```
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full,
    test_size=0.33, random_state=1)
# summarize data split
print('Train: %s, Val: %s, Test: %s' % (X_train.shape, X_val.shape, X_test.shape))
# create the base models
models = get_models()
# train the blending ensemble
blender = fit_ensemble(models, X_train, X_val, y_train, y_val)
# make predictions on test set
yhat = predict_ensemble(models, blender, X_test)
# evaluate predictions
score = mean_absolute_error(y_test, yhat)
print('Blending MAE: %.3f' % score)
```

Listing 29.28: Example of evaluating the blending ensemble on the synthetic regression dataset.

Running the example first reports the shape of the train, validation, and test datasets, then the MAE of the ensemble on the test dataset.

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

In this case, we can see that the blending ensemble achieved a MAE of about 0.237 on the test dataset.

```
Train: (3350, 20), Val: (1650, 20), Test: (5000, 20)
Blending MAE: 0.237
```

Listing 29.29: Example output from evaluating the blending ensemble on the synthetic regression dataset.

As with classification, the blending ensemble is only useful if it performs better than any of the base models that contribute to the ensemble. We can check this by evaluating each base-model in isolation by first fitting it on the entire training dataset (unlike the blending ensemble) and making predictions on the test dataset (like the blending ensemble). The example below evaluates each of the base models in isolation on the synthetic regression predictive modeling dataset.

```
# evaluate base models in isolation on the regression dataset
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR

# get the dataset
def get_dataset():
  X, y = make_regression(n_samples=10000, n_features=20, n_informative=10, noise=0.3,
      random_state=7)
  return X, y

# get a list of base models
```

```
def get_models():
  models = list()
  models.append(('lr', LinearRegression()))
  models.append(('knn', KNeighborsRegressor()))
  models.append(('cart', DecisionTreeRegressor()))
  models.append(('svm', SVR()))
  return models

# define dataset
X, y = get_dataset()
# split dataset into train and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.5,
    random_state=1)
# summarize data split
print('Train: %s, Test: %s' % (X_train_full.shape, X_test.shape))
# create the base models
models = get_models()
# evaluate standalone model
for name, model in models:
  # fit the model on the training dataset
  model.fit(X_train_full, y_train_full)
  # make a prediction on the test dataset
  yhat = model.predict(X_test)
  # evaluate the predictions
  score = mean_absolute_error(y_test, yhat)
  # report the score
  print('>%s MAE: %.3f' % (name, score))
```

Listing 29.30: Example of evaluating standalone models on the synthetic regression dataset.

Running the example first reports the shape of the full train and test datasets, then the MAE of each base-model on the test dataset.

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

In this case, we can see that indeed the linear regression model has performed slightly better than the blending ensemble, achieving a MAE of 0.236 as compared to 0.237 with the ensemble. This may be because of the way that the synthetic dataset was constructed. Nevertheless, in this case, we would choose to use the linear regression model directly on this problem. This highlights the importance of checking the performance of the contributing models before adopting an ensemble model as the final model.

```
Train: (5000, 20), Test: (5000, 20)
>lr MAE: 0.236
>knn MAE: 100.169
>cart MAE: 133.744
>svm MAE: 138.195
```

Listing 29.31: Example output from evaluating standalone models on the synthetic regression dataset.

Again, we may choose to use a blending ensemble as our final model for regression. This involves fitting splitting the entire dataset into train and validation sets to fit the base and

meta-models respectively, then the ensemble can be used to make a prediction for a new row of data. The complete example of making a prediction on new data with a blending ensemble for regression is listed below.

```
# example of making a prediction with a blending ensemble for regression
from numpy import hstack
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR

# get the dataset
def get_dataset():
  X, y = make_regression(n_samples=10000, n_features=20, n_informative=10, noise=0.3,
      random_state=7)
  return X, y

# get a list of base models
def get_models():
  models = list()
  models.append(('lr', LinearRegression()))
  models.append(('knn', KNeighborsRegressor()))
  models.append(('cart', DecisionTreeRegressor()))
  models.append(('svm', SVR()))
  return models

# fit the blending ensemble
def fit_ensemble(models, X_train, X_val, y_train, y_val):
  # fit all models on the training set and predict on hold out set
  meta_X = list()
  for _, model in models:
    # fit in training set
    model.fit(X_train, y_train)
    # predict on hold out set
    yhat = model.predict(X_val)
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
    # store predictions as input for blending
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # define blending model
  blender = LinearRegression()
  # fit on predictions from base models
  blender.fit(meta_X, y_val)
  return blender

# make a prediction with the blending ensemble
def predict_ensemble(models, blender, X_test):
  # make predictions with base models
  meta_X = list()
  for _, model in models:
    # predict with base model
    yhat = model.predict(X_test)
```

```
    # reshape predictions into a matrix with one column
    yhat = yhat.reshape(len(yhat), 1)
    # store prediction
    meta_X.append(yhat)
  # create 2d array from predictions, each set is an input feature
  meta_X = hstack(meta_X)
  # predict
  return blender.predict(meta_X)

# define dataset
X, y = get_dataset()
# split dataset set into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.33, random_state=1)
# summarize data split
print('Train: %s, Val: %s' % (X_train.shape, X_val.shape))
# create the base models
models = get_models()
# train the blending ensemble
blender = fit_ensemble(models, X_train, X_val, y_train, y_val)
# make a prediction on a new row of data
row = [-0.24038754, 0.55423865, -0.48979221, 1.56074459, -1.16007611, 1.10049103,
    1.18385406, -1.57344162, 0.97862519, -0.03166643, 1.77099821, 1.98645499, 0.86780193,
    2.01534177, 2.51509494, -1.04609004, -0.19428148, -0.05967386, -2.67168985, 1.07182911]
yhat = predict_ensemble(models, blender, [row])
# summarize prediction
print('Predicted: %.3f' % (yhat[0]))
```

Listing 29.32: Example of making a prediction with a blending ensemble for regression.

Running the example fits the blending ensemble model on the dataset and is then used to make a prediction on a new row of data, as we might when using the model in an application.

```
Train: (6700, 20), Val: (3300, 20)
Predicted: 359.986
```

Listing 29.33: Example output from making a prediction with a blending ensemble for regression.

29.6 Further Reading

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

Papers

- *Feature-Weighted Linear Stacking*, 2009.
  https://arxiv.org/abs/0911.0460
- *The BellKor 2008 Solution to the Netflix Prize*, 2008.
  https://netflixprize.com/assets/ProgressPrize2008_BellKor.pdf
- *Kaggle Ensemble Guide*, MLWave, 2015.
  https://mlwave.com/kaggle-ensembling-guide/

Articles

- Netflix Prize, Wikipedia.
  https://en.wikipedia.org/wiki/Netflix_Prize

29.7 Summary

In this tutorial, you discovered how to develop and evaluate a blending ensemble in Python. Specifically, you learned:

- Blending ensembles are a type of stacking where the meta-model is fit using predictions on a holdout validation dataset instead of out-of-fold predictions.
- How to develop a blending ensemble, including functions for training the model and making predictions on new data.
- How to evaluate blending ensembles for classification and regression predictive modeling problems.

Next

In the next section, we will take a closer look at an extension to stacking called the super learner ensemble.

Chapter 30

Super Learner Ensemble

Selecting a machine learning algorithm for a predictive modeling problem involves evaluating many different models and model configurations using k-fold cross-validation. The super learner is an ensemble machine learning algorithm that combines all of the models and model configurations that you might investigate for a predictive modeling problem and uses them to make a prediction as-good-as or better than any single model that you may have investigated. The super learner algorithm is an application of stacked generalization (stacking) to k-fold cross-validation where all models use the same k-fold splits of the data and a meta-model is fit on the out-of-fold predictions from each model. In this tutorial, you will discover the super learner ensemble machine learning algorithm. After completing this tutorial, you will know:

- Super learner is the application of stacked generalization using out-of-fold predictions during k-fold cross-validation.
- The super learner ensemble algorithm is straightforward to implement in Python using scikit-learn models.
- The ML-Ensemble library provides a convenient implementation that allows the super learner to be fit and used in just a few lines of code.

Let's get started.

30.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Super Learner Ensemble
2. Develop Super Learner Ensembles
3. Evaluate Super Learner Ensembles

30.2 Super Learner Ensemble

There are many hundreds of models to choose from for a predictive modeling problem; which one is best? Then, after a model is chosen, how do you best configure it for your specific dataset? These are open questions in applied machine learning. The best answer we have at the moment is to use empirical experimentation to test and discover what works best for your dataset.

> In practice, it is generally impossible to know a priori which learner will perform best for a given prediction problem and data set.
>
> — *Super Learner*, 2007.

This involves selecting many different algorithms that may be appropriate for your regression or classification problem and evaluating their performance on your dataset using a resampling technique, such as k-fold cross-validation. The algorithm that performs the best on your dataset according to k-fold cross-validation is then selected, fit on all available data, and you can then start using it to make predictions. There is an alternative approach. Consider that you have already fit many different algorithms on your dataset, and some algorithms have been evaluated many times with different configurations. You may have many tens or hundreds of different models of your problem. Why not use all those models instead of the best model from the group? This is the intuition behind the so-called *super learner* ensemble algorithm.

The super learner algorithm involves first pre-defining the k-fold splits of your data, then evaluating all different algorithms and algorithm configurations on the same splits of the data. All out-of-fold predictions are then kept and used to train a model that learns how to best combine the predictions.

> The algorithms may differ in the subset of the covariates used, the basis functions, the loss functions, the searching algorithm, and the range of tuning parameters, among others.
>
> — *Super Learner In Prediction*, 2010.

The results of this model should be no worse than the best performing model evaluated during k-fold cross-validation and has the likelihood of performing better than any single model. The super learner algorithm was proposed by Mark van der Laan, et al. from Berkeley in their 2007 paper titled *Super Learner*. It was published in a biological journal, which may be sheltered from the broader machine learning community. The super learner technique is an example of the general method called *stacked generalization*, or *stacking* for short (introduced in Chapter 28).

> The super learner is related to the stacking algorithm introduced in neural networks context ...
>
> — *Super Learner In Prediction*, 2010.

We can think of the Super Learner Ensemble as a specific configuration of stacking specifically to k-fold cross-validation with all models considered for a predictive modeling project. The procedure can be summarized as follows:

1. Select a k-fold split of the training dataset.

2. Select m base-models or model configurations.
3. For each base-model:
   (a) Evaluate using k-fold cross-validation.
   (b) Store all out-of-fold predictions.
   (c) Fit the model on the full training dataset and store.
4. Fit a meta-model on the out-of-fold predictions.
5. Evaluate the model on a holdout dataset or use model to make predictions.

The image below, taken from the original paper, summarizes this data flow.

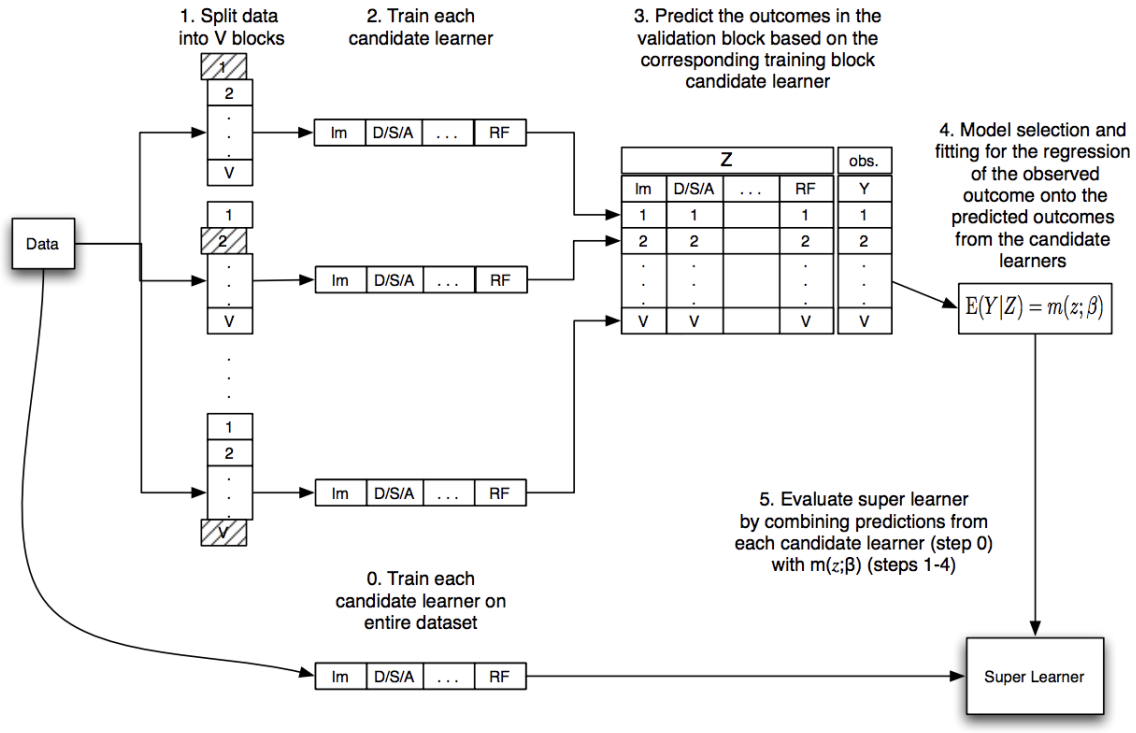

Figure 30.1: Diagram Showing the Data Flow of the Super Learner Algorithm. Taken from *Super Learner*.

Now that we are familiar with the super learner algorithm, let's look at a worked example.

30.3 Develop Super Learner Ensembles

The Super Learner algorithm is relatively straightforward to implement on top of the scikit-learn Python machine learning library. In fact we could just use the `StackingRegressor` and `StackingClassifier` classes (see Chapter 28). Nevertheless, in this section, we will develop an example of super learning for both classification and regression that you can adapt to your own problems.

30.3.1 Super Learner for Classification

The inputs to the meta learner can be class labels or class probabilities, with the latter more likely to be useful given the increased granularity or uncertainty captured in the predictions. First, we can use the `make_classification()` function to create a synthetic binary classification problem with 1,000 examples and 20 input features. The complete example is listed below.

```
# synthetic binary 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=2)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 30.1: Example of creating the synthetic classification dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

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

Listing 30.2: Example output from creating the synthetic classification dataset.

Next, we will define a bunch of different classification models. In this case, we will use nine different algorithms with modest configurations. You can use any models or model configurations you like. The `get_models()` function below defines all of the models and returns them as a list.

```
# create a list of base-models
def get_models():
  models = list()
  models.append(LogisticRegression(solver='liblinear'))
  models.append(DecisionTreeClassifier())
  models.append(SVC(gamma='scale', probability=True))
  models.append(GaussianNB())
  models.append(KNeighborsClassifier())
  models.append(AdaBoostClassifier())
  models.append(BaggingClassifier(n_estimators=10))
  models.append(RandomForestClassifier(n_estimators=10))
  models.append(ExtraTreesClassifier(n_estimators=10))
  return models
```

Listing 30.3: Example of defining base classification models to use in the super learner ensemble.

Next, we will use k-fold cross-validation to make out-of-fold predictions that will be used as the dataset to train the meta-model or super learner. This involves first splitting the data into k folds; we will use 10. For each fold, we will fit the model on the training part of the split and make out-of-fold predictions on the test part of the split. This is repeated for each model and all out-of-fold predictions are stored.

Each out-of-fold prediction will be a column for the meta-model input. We will collect columns from each algorithm for one fold of the data, horizontally stacking the rows. Then for all groups of columns we collect, we will vertically stack these rows into one long dataset with 500 rows and nine columns. The `get_out_of_fold_predictions()` function below does this for a given test dataset and list of models; it will return the input and output dataset required to train the meta-model.

```
# collect out of fold predictions from cross-validation
def get_out_of_fold_predictions(X, y, models):
  meta_X, meta_y = list(), list()
  # define split of data
  kfold = KFold(n_splits=10, shuffle=True)
  # enumerate splits
  for train_ix, test_ix in kfold.split(X):
    fold_yhats = list()
    # get data
    train_X, test_X = X[train_ix], X[test_ix]
    train_y, test_y = y[train_ix], y[test_ix]
    meta_y.extend(test_y)
    # fit and make predictions with each sub-model
    for model in models:
      model.fit(train_X, train_y)
      yhat = model.predict_proba(test_X)
      # store columns
      fold_yhats.append(yhat)
    # store fold yhats as columns
    meta_X.append(hstack(fold_yhats))
  return vstack(meta_X), asarray(meta_y)
```

Listing 30.4: Example of a function for making out of fold predictions.

We can then call the function to get the models and the function to prepare the meta-model dataset.

```
...
# get models
models = get_models()
# get out of fold predictions
meta_X, meta_y = get_out_of_fold_predictions(X, y, models)
print('Meta ', meta_X.shape, meta_y.shape)
```

Listing 30.5: Example of a preparing the dataset for the meta-model.

Next, we can fit all of the base-models on the entire training dataset.

```
# fit all base models on the training dataset
def fit_base_models(X, y, models):
  for model in models:
    model.fit(X, y)
```

Listing 30.6: Example of a function for fitting the base models.

Then, we can fit the meta-model on the prepared dataset. In this case, we will use a logistic regression model as the meta-model.

```
# fit a meta-model
def fit_meta_model(X, y):
  model = LogisticRegression(solver='liblinear')
  model.fit(X, y)
  return model
```

Listing 30.7: Example of a function for fitting the meta-model.

And classification accuracy will be used to report model performance. The complete example of the super learner algorithm for classification using scikit-learn models is listed below.

```
# example of a super learner model for binary classification
from numpy import hstack
from numpy import vstack
from numpy import asarray
from sklearn.datasets import make_classification
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier

# create a list of base-models
def get_models():
  models = list()
  models.append(LogisticRegression(solver='liblinear'))
  models.append(DecisionTreeClassifier())
  models.append(SVC(gamma='scale', probability=True))
  models.append(GaussianNB())
  models.append(KNeighborsClassifier())
  models.append(AdaBoostClassifier())
  models.append(BaggingClassifier(n_estimators=10))
  models.append(RandomForestClassifier(n_estimators=10))
  models.append(ExtraTreesClassifier(n_estimators=10))
  return models

# collect out of fold predictions from cross validation
def get_out_of_fold_predictions(X, y, models):
  meta_X, meta_y = list(), list()
  # define split of data
  kfold = KFold(n_splits=10, shuffle=True)
  # enumerate splits
  for train_ix, test_ix in kfold.split(X):
    fold_yhats = list()
    # get data
    train_X, test_X = X[train_ix], X[test_ix]
    train_y, test_y = y[train_ix], y[test_ix]
    meta_y.extend(test_y)
    # fit and make predictions with each sub-model
    for model in models:
      model.fit(train_X, train_y)
      yhat = model.predict_proba(test_X)
      # store columns
      fold_yhats.append(yhat)
    # store fold yhats as columns
    meta_X.append(hstack(fold_yhats))
  return vstack(meta_X), asarray(meta_y)

# fit all base models on the training dataset
```

```
def fit_base_models(X, y, models):
  for model in models:
    model.fit(X, y)

# fit a meta model
def fit_meta_model(X, y):
  model = LogisticRegression(solver='liblinear')
  model.fit(X, y)
  return model

# evaluate a list of models on a dataset
def evaluate_models(X, y, models):
  for model in models:
    yhat = model.predict(X)
    score = accuracy_score(y, yhat)
    print('%s: %.3f' % (model.__class__.__name__, score))

# make predictions with stacked model
def super_learner_predictions(X, models, meta_model):
  meta_X = list()
  for model in models:
    yhat = model.predict_proba(X)
    meta_X.append(yhat)
  meta_X = hstack(meta_X)
  # predict
  return meta_model.predict(meta_X)

# create the inputs and outputs
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=2)
# split
X, X_val, y, y_val = train_test_split(X, y, test_size=0.50)
print('Train', X.shape, y.shape, 'Test', X_val.shape, y_val.shape)
# get models
models = get_models()
# get out of fold predictions
meta_X, meta_y = get_out_of_fold_predictions(X, y, models)
print('Meta ', meta_X.shape, meta_y.shape)
# fit base models
fit_base_models(X, y, models)
# fit the meta model
meta_model = fit_meta_model(meta_X, meta_y)
# evaluate base models
evaluate_models(X_val, y_val, models)
# evaluate meta model
yhat = super_learner_predictions(X_val, models, meta_model)
score = accuracy_score(y_val, yhat)
print('Super Learner: %.3f' % score)
```

Listing 30.8: Example of evaluating a super learner ensemble on the synthetic classification dataset.

The shape of the dataset and the prepared meta dataset is reported, followed by the performance of the base-models on the holdout dataset and finally the super model itself on the holdout dataset.

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

In this case, we can see that the super learner has better performance than the base learner algorithms, achieving a classification accuracy of about 90 percent.

```
Train (500, 20) (500,) Test (500, 20) (500,)
Meta (500, 18) (500,)

LogisticRegression: 0.860
DecisionTreeClassifier: 0.772
SVC: 0.888
GaussianNB: 0.832
KNeighborsClassifier: 0.854
AdaBoostClassifier: 0.838
BaggingClassifier: 0.824
RandomForestClassifier: 0.850
ExtraTreesClassifier: 0.842

Super Learner: 0.902
```

Listing 30.9: Example output from evaluating a super learner ensemble on the synthetic classification dataset.

Now that we have seen how to develop a super learner for classification, let's look at an example for regression.

30.3.2 Super Learner for Regression

In this section, we will develop a super learner ensemble for a regression predictive modeling problem. First, we can use the `make_regression()` function to create a synthetic regression problem with 1,000 examples and 100 input features. The complete example is listed below.

```
# synthetic regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=100, noise=0.5, random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

Listing 30.10: Example of creating the synthetic regression dataset.

Running the example creates the dataset and summarizes the shape of the input and output components.

```
(1000, 100) (1000,)
```

Listing 30.11: Example output from creating the synthetic regression dataset.

Next, we will define a bunch of different regression models. In this case, we will use eight different algorithms with modest configurations. You can use any models or model configurations you like. Note, we have explicitly excluded the linear regression model from this ensemble as it is used in the generation of the synthetic dataset. The `get_models()` function below defines all of the models and returns them as a list.

```
# create a list of base-models
def get_models():
  models = list()
  models.append(ElasticNet())
  models.append(SVR(gamma='scale'))
  models.append(DecisionTreeRegressor())
  models.append(KNeighborsRegressor())
  models.append(AdaBoostRegressor())
  models.append(BaggingRegressor(n_estimators=10))
  models.append(RandomForestRegressor(n_estimators=10))
  models.append(ExtraTreesRegressor(n_estimators=10))
  return models
```

Listing 30.12: Example of defining base regression models to use in the super learner ensemble.

Next, we can change the `get_out_of_fold_predictions()` function to predict numerical values by a call to the `predict()` function.

```
# collect out of fold predictions from cross-validation
def get_out_of_fold_predictions(X, y, models):
  meta_X, meta_y = list(), list()
  # define split of data
  kfold = KFold(n_splits=10, shuffle=True)
  # enumerate splits
  for train_ix, test_ix in kfold.split(X):
    fold_yhats = list()
    # get data
    train_X, test_X = X[train_ix], X[test_ix]
    train_y, test_y = y[train_ix], y[test_ix]
    meta_y.extend(test_y)
    # fit and make predictions with each sub-model
    for model in models:
      model.fit(train_X, train_y)
      yhat = model.predict(test_X)
      # store columns
      fold_yhats.append(yhat.reshape(len(yhat),1))
    # store fold yhats as columns
    meta_X.append(hstack(fold_yhats))
  return vstack(meta_X), asarray(meta_y)
```

Listing 30.13: Example of a function for making out of fold predictions.

Then, we can fit the meta-model on the prepared dataset. In this case, we will use a linear regression model as the meta-model, as was used in the original paper.

```
# fit a meta-model
def fit_meta_model(X, y):
  model = LinearRegression()
  model.fit(X, y)
  return model
```

Listing 30.14: Example of a function for fitting the meta-model.

Next, we can evaluate the base-models on the holdout dataset and report the mean absolute error (MAE).

```
# evaluate a list of models on a dataset
```

```
def evaluate_models(X, y, models):
  for model in models:
    yhat = model.predict(X)
    mae = mean_absolute_error(y, yhat)
    print('%s: MAE %.3f' % (model.__class__.__name__, mae))
```

Listing 30.15: Example of a function for evaluating models on a hold out dataset.

And, finally, use the super learner (base and meta-model) to make predictions on the holdout dataset and evaluate the performance of the approach. The `super_learner_predictions()` function below will use the meta-model to make predictions for new data.

```
# make predictions with stacked model
def super_learner_predictions(X, models, meta_model):
  meta_X = list()
  for model in models:
    yhat = model.predict(X)
    meta_X.append(yhat.reshape(len(yhat),1))
  meta_X = hstack(meta_X)
  # predict
  return meta_model.predict(meta_X)
```

Listing 30.16: Example of a function for making predictions the super learner ensemble.

We can call this function and evaluate the predictions, reporting the ensemble mean absolute error.

```
...
# evaluate meta model
yhat = super_learner_predictions(X_val, models, meta_model)
score = mean_absolute_error(y_val, yhat)
print('Super Learner: MAE %.3f' % score)
```

Listing 30.17: Example of summarizing the performance of the super learner ensemble.

Tying this all together, the complete example of a super learner algorithm for regression using scikit-learn models is listed below.

```
# example of a super learner model for regression
from numpy import hstack
from numpy import vstack
from numpy import asarray
from sklearn.datasets import make_regression
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import ElasticNet
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import BaggingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor

# create a list of base-models
```

```
def get_models():
  models = list()
  models.append(ElasticNet())
  models.append(SVR(gamma='scale'))
  models.append(DecisionTreeRegressor())
  models.append(KNeighborsRegressor())
  models.append(AdaBoostRegressor())
  models.append(BaggingRegressor(n_estimators=10))
  models.append(RandomForestRegressor(n_estimators=10))
  models.append(ExtraTreesRegressor(n_estimators=10))
  return models

# collect out of fold predictions from cross validation
def get_out_of_fold_predictions(X, y, models):
  meta_X, meta_y = list(), list()
  # define split of data
  kfold = KFold(n_splits=10, shuffle=True)
  # enumerate splits
  for train_ix, test_ix in kfold.split(X):
    fold_yhats = list()
    # get data
    train_X, test_X = X[train_ix], X[test_ix]
    train_y, test_y = y[train_ix], y[test_ix]
    meta_y.extend(test_y)
    # fit and make predictions with each sub-model
    for model in models:
      model.fit(train_X, train_y)
      yhat = model.predict(test_X)
      # store columns
      fold_yhats.append(yhat.reshape(len(yhat),1))
    # store fold yhats as columns
    meta_X.append(hstack(fold_yhats))
  return vstack(meta_X), asarray(meta_y)

# fit all base models on the training dataset
def fit_base_models(X, y, models):
  for model in models:
    model.fit(X, y)

# fit a meta model
def fit_meta_model(X, y):
  model = LinearRegression()
  model.fit(X, y)
  return model

# evaluate a list of models on a dataset
def evaluate_models(X, y, models):
  for model in models:
    yhat = model.predict(X)
    mae = mean_absolute_error(y, yhat)
    print('%s: MAE %.3f' % (model.__class__.__name__, mae))

# make predictions with stacked model
def super_learner_predictions(X, models, meta_model):
  meta_X = list()
  for model in models:
```

```
    yhat = model.predict(X)
    meta_X.append(yhat.reshape(len(yhat),1))
  meta_X = hstack(meta_X)
  # predict
  return meta_model.predict(meta_X)

# create the inputs and outputs
X, y = make_regression(n_samples=1000, n_features=100, noise=0.5, random_state=1)
# split
X, X_val, y, y_val = train_test_split(X, y, test_size=0.50)
print('Train', X.shape, y.shape, 'Test', X_val.shape, y_val.shape)
# get models
models = get_models()
# get out of fold predictions
meta_X, meta_y = get_out_of_fold_predictions(X, y, models)
print('Meta ', meta_X.shape, meta_y.shape)
# fit base models
fit_base_models(X, y, models)
# fit the meta model
meta_model = fit_meta_model(meta_X, meta_y)
# evaluate base models
evaluate_models(X_val, y_val, models)
# evaluate meta model
yhat = super_learner_predictions(X_val, models, meta_model)
score = mean_absolute_error(y_val, yhat)
print('Super Learner: MAE %.3f' % score)
```

Listing 30.18: Example of evaluating a super learner ensemble on the synthetic regression dataset.

Running the example first reports the shape of the prepared dataset, then the shape of the dataset for the meta-model. Next, the performance of each base-model is reported on the holdout dataset, and finally, the performance of the super learner on the holdout dataset.

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

In this case, we can see that the best performing standalone model is the ElasticNet with a MAE of about 61. We can see that the super learner ensemble out-performs all standalone models achieving a MAE of about 24.

```
Train (500, 100) (500,) Test (500, 100) (500,)
Meta  (500, 9) (500,)

ElasticNet: MAE 61.527
SVR: MAE 164.208
DecisionTreeRegressor: MAE 148.379
KNeighborsRegressor: MAE 139.828
AdaBoostRegressor: MAE 99.176
BaggingRegressor: MAE 107.717
RandomForestRegressor: MAE 106.724
ExtraTreesRegressor: MAE 106.967

Super Learner: MAE 24.180
```

Listing 30.19: Example output from evaluating a super learner ensemble on the synthetic regression dataset.

Next, let's take a look at how we might use a third-party library to implement the super learner ensemble.

30.4 Evaluate Super Learner Ensembles

Implementing the super learner manually is a good exercise but is not ideal. We may introduce bugs in the implementation and the example as listed does not make use of multiple cores to speed up the execution. Thankfully, Sebastian Flennerhag provides an efficient and tested implementation of the Super Learner algorithm and other ensemble algorithms in his ML-Ensemble (mlens) Python library. It is specifically designed to work with scikit-learn models. First, the library must be installed, which can be achieved via `pip`, as follows:

```
pip install mlens
```

Listing 30.20: Example of installing the mlens library with `pip`.

You can then confirm that the mlens library was installed correctly and can be used by running the following script.

```
# check the version of the mlens library
import mlens
# report library version
print(mlens.__version__)
```

Listing 30.21: Example of checking the mlens library version.

Running the script will print your version of the mlens library you have installed. Your version should be the same or higher. If not, you must upgrade your version of the mlens library.

```
0.2.3
```

Listing 30.22: Example output from checking the mlens library version.

Next, a `SuperLearner` class can be defined, models added via a call to the `add()` function, the meta learner added via a call to the `add_meta()` function, then the model used like any other scikit-learn model.

```
...
# configure model
ensemble = SuperLearner(...)
# add list of base learners
ensemble.add(...)
# add meta learner
ensemble.add_meta(...)
# use model ...
```

Listing 30.23: Example of defining a super learner ensemble using the mlens library.

We can use this class on the classification and regression problems from the previous section.

30.4.1 Super Learner for Classification

In this section we will explore how to develop a super learner ensemble for classification using the mlens library. First, we will define the super learner to evaluate base learners using classification accuracy with 10-fold cross-validation on the entire training dataset.

```
...
# create the super learner
ensemble = SuperLearner(scorer=accuracy_score, folds=10, shuffle=True,
    sample_size=len(X_train))
```

Listing 30.24: Example of defining a super learner for classification.

We will use the same nine models for classification as we did in the previous section, provided a list from the `get_models()` function.

```
# create a list of base-models
def get_models():
  models = list()
  models.append(LogisticRegression(solver='liblinear'))
  models.append(DecisionTreeClassifier())
  models.append(SVC(gamma='scale', probability=True))
  models.append(GaussianNB())
  models.append(KNeighborsClassifier())
  models.append(AdaBoostClassifier())
  models.append(BaggingClassifier(n_estimators=10))
  models.append(RandomForestClassifier(n_estimators=10))
  models.append(ExtraTreesClassifier(n_estimators=10))
  return models
```

Listing 30.25: Example of defining a base models for classification.

These models can then be added to the super learner ensemble.

```
...
# add the base models
ensemble.add(get_models())
```

Listing 30.26: Example of adding base models to the super learner ensemble for classification.

Next, the meta-model can be specified, in this case a logistic regression model with a stable solver.

```
...
# add the meta model
ensemble.add_meta(LogisticRegression(solver='lbfgs'))
```

Listing 30.27: Example of adding meta-model to the super learner ensemble for classification.

We can then fit the super learner ensemble on the entire training dataset and report the performance of each base-model.

```
...
# fit the super learner
ensemble.fit(X_train, y_train)
# summarize base learners
print(ensemble.data)
```

Listing 30.28: Example of fitting and reporting the performance the super learner ensemble on the training dataset.

Finally, we can evaluate the performance of the ensemble on the hold out test dataset.

```
...
# make predictions on hold out set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Super Learner Accuracy: %.3f' % score)
```

Listing 30.29: Example of evaluating the performance of the super learner ensemble for classification.

Tying this together, the complete example of evaluating a super learner ensemble of the synthetic binary classification task is listed below.

```
# example of evaluating a super learner ensemble for classification with the mlens library
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from mlens.ensemble import SuperLearner

# create a list of base-models
def get_models():
  models = list()
  models.append(LogisticRegression(solver='liblinear'))
  models.append(DecisionTreeClassifier())
  models.append(SVC(gamma='scale', probability=True))
  models.append(GaussianNB())
  models.append(KNeighborsClassifier())
  models.append(AdaBoostClassifier())
  models.append(BaggingClassifier(n_estimators=10))
  models.append(RandomForestClassifier(n_estimators=10))
  models.append(ExtraTreesClassifier(n_estimators=10))
  return models

# create the inputs and outputs
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
    random_state=2)
# split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
# create the super learner
ensemble = SuperLearner(scorer=accuracy_score, folds=10, shuffle=True,
    sample_size=len(X_train))
# add the base models
ensemble.add(get_models())
# add the meta model
ensemble.add_meta(LogisticRegression(solver='lbfgs'))
# fit the super learner
```

```
ensemble.fit(X_train, y_train)
# summarize base learners
print(ensemble.data)
# make predictions on hold out set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = accuracy_score(y_test, yhat)
print('Super Learner Accuracy: %.3f' % score)
```

Listing 30.30: Example of evaluating an mlens super learner ensemble on the synthetic classification dataset.

Running the example summarizes the performance of the base-models on the training dataset, and finally the performance of the super learner on the holdout dataset.

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

In this case, we can see that the super learner performs well on the dataset achieving a classification accuracy of about 90 percent. More importantly, the model is fit and evaluated very quickly as compared to the manual example in the previous section. Note that we cannot compare the base learner scores in the table to the super learner as the base learners were evaluated on the training dataset only, not the holdout dataset.

```
                              score-m score-s ft-m  ft-s  pt-m  pt-s
layer-1 adaboostclassifier       0.84    0.04  0.53  0.03  0.01  0.00
layer-1 baggingclassifier        0.84    0.05  0.16  0.01  0.00  0.00
layer-1 decisiontreeclassifier   0.75    0.06  0.02  0.01  0.00  0.00
layer-1 extratreesclassifier     0.81    0.04  0.11  0.02  0.00  0.00
layer-1 gaussiannb               0.81    0.07  0.01  0.00  0.00  0.00
layer-1 kneighborsclassifier     0.88    0.03  0.00  0.00  0.04  0.01
layer-1 logisticregression       0.84    0.03  0.00  0.00  0.00  0.00
layer-1 randomforestclassifier   0.82    0.04  0.10  0.03  0.00  0.00
layer-1 svc                      0.91    0.03  0.02  0.00  0.00  0.00

Super Learner Accuracy: 0.906
```

Listing 30.31: Example output from evaluating an mlens super learner ensemble on the synthetic classification dataset.

Next, let's explore how we might evaluate a super learner ensemble for regression.

30.4.2 Super Learner for Regression

In this section we will explore how to develop a super learner ensemble for regression using the mlens library. First, we can configure the `SuperLearner` with 10-fold cross-validation, mean absolute error evaluation function, and the use of the entire training dataset when preparing out-of-fold predictions to use as input for the meta-model.

```
...
# create the super learner
ensemble = SuperLearner(scorer=mean_absolute_error, folds=10, shuffle=True,
    sample_size=len(X_train))
```

```
# add the base models
ensemble.add(get_models())
# add the meta model
ensemble.add_meta(LinearRegression())
```

Listing 30.32: Example of defining the super learner ensemble for regression.

We can then fit the model on the training dataset.

```
...
# fit the super learner
ensemble.fit(X, y)
```

Listing 30.33: Example of fitting the super learner ensemble.

Once fit, we can get a nice report of the performance of each of the base-models on the training dataset using k-fold cross-validation by accessing the `data` attribute on the model.

```
...
# summarize base learners
print(ensemble.data)
```

Listing 30.34: Example of accessing a report on the super learner ensemble's performance.

And that's all there is to it. Tying this together, the complete example of evaluating a super learner using the mlens library for regression is listed below.

```
# example of evaluating a super learner ensemble for regression with the mlens library
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import ElasticNet
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import BaggingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor
from mlens.ensemble import SuperLearner

# create a list of base-models
def get_models():
  models = list()
  models.append(ElasticNet())
  models.append(SVR(gamma='scale'))
  models.append(DecisionTreeRegressor())
  models.append(KNeighborsRegressor())
  models.append(AdaBoostRegressor())
  models.append(BaggingRegressor(n_estimators=10))
  models.append(RandomForestRegressor(n_estimators=10))
  models.append(ExtraTreesRegressor(n_estimators=10))
  return models

# create the inputs and outputs
X, y = make_regression(n_samples=1000, n_features=100, noise=0.5, random_state=1)
# split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
# create the super learner
ensemble = SuperLearner(scorer=mean_absolute_error, folds=10, shuffle=True,
    sample_size=len(X_train))
# add the base models
ensemble.add(get_models())
# add the meta model
ensemble.add_meta(LinearRegression())
# fit the super learner
ensemble.fit(X_train, y_train)
# summarize base learners
print(ensemble.data)
# make predictions on hold out set
yhat = ensemble.predict(X_test)
# evaluate predictions
score = mean_absolute_error(y_test, yhat)
print('Super Learner MAE: %.3f' % score)
```

Listing 30.35: Example of evaluating an mlens super learner ensemble on the synthetic regression dataset.

Running the example first reports the MAE for (score-m) for each base-model, then reports the MAE for the super learner itself. Fitting and evaluating is very fast given the use of multi-threading in the backend allowing all cores of your machine to be used.

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

In this case, we can see that the super learner ensemble achieves a MAE of about 20. Note that we cannot compare the base learner scores in the table to the super learner as the base learners were evaluated on the training dataset only, not the holdout dataset.

```
                              score-m score-s ft-m ft-s pt-m pt-s
layer-1 adaboostregressor       94.55   10.59 0.61 0.02 0.04 0.01
layer-1 baggingregressor       107.79    9.55 0.25 0.01 0.01 0.00
layer-1 decisiontreeregressor  160.03   20.10 0.04 0.01 0.00 0.00
layer-1 elasticnet              59.51    6.00 0.01 0.00 0.00 0.00
layer-1 extratreesregressor    105.80   10.01 0.14 0.01 0.01 0.01
layer-1 kneighborsregressor    148.74   11.87 0.00 0.00 0.01 0.00
layer-1 randomforestregressor  106.13    9.68 0.22 0.03 0.00 0.00
layer-1 svr                    166.55   15.35 0.03 0.00 0.00 0.00

Super Learner MAE: 20.335
```

Listing 30.36: Example output from evaluating an mlens super learner ensemble on the synthetic regression dataset.

30.5 Further Reading

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

Books

- *Targeted Learning: Causal Inference for Observational and Experimental Data*, 2011.
  https://amzn.to/30aiIx1
- *Targeted Learning in Data Science: Causal Inference for Complex Longitudinal Studies*, 2018.
  https://amzn.to/3099cKm

Papers

- *Super Learner*, 2007.
  https://www.degruyter.com/view/j/sagmb.2007.6.issue-1/sagmb.2007.6.1.1309/sagmb.2007.6.1.1309.xml
- *Super Learner In Prediction*, 2010.
  https://biostats.bepress.com/ucbbiostat/paper266/
- *Super Learning*, 2011.
  https://link.springer.com/chapter/10.1007/978-1-4419-9782-1_3
- *Super Learning*, Slides.
  https://vanderlaan-lab.org/teach-files/BASS_sec1_3.1.pdf

R Software

- SuperLearner: Super Learner Prediction, CRAN.
  https://cran.r-project.org/web/packages/SuperLearner/
- SuperLearner: Prediction model ensembling method, GitHub.
  https://github.com/ecpolley/SuperLearner
- *Guide to SuperLearner*, Vignette, 2017.
  https://cran.r-project.org/web/packages/SuperLearner/vignettes/Guide-to-SuperLearner.html

Python Software

- ML-Ensemble, Homepage.
  http://ml-ensemble.com/
- ML-ENS Documentation.
  http://ml-ensemble.com/info/tutorials/start.html#ensemble-guide
- ML-ENS, GitHub.
  https://github.com/flennerhag/mlens

30.6 Summary

In this tutorial, you discovered the super learner ensemble machine learning algorithm. Specifically, you learned:

- Super learner is the application of stacked generalization using out-of-fold predictions during k-fold cross-validation.
- The super learner ensemble algorithm is straightforward to implement in Python using scikit-learn models.
- The ML-Ensemble library provides a convenient implementation that allows the super learner to be fit and used in just a few lines of code.

Next

This was the final tutorial in this part, in the next part we will review useful resources for learning more about ensemble algorithms.

Part VIII

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with ensemble learning. As you start to work on projects and expand your existing knowledge of the techniques, you may need help. This appendix points out some of the best sources to turn to.

A.1 Ensemble Learning Books

There are very few books dedicated to the topic of ensemble learning. Books that I would recommend include:

- *Ensemble Methods: Foundations and Algorithms*, 2012.
  https://amzn.to/2XZzrjG
- *Ensemble Learning: Pattern Classification Using Ensemble Methods*, 2019.
  https://amzn.to/2XYTor0
- *Pattern Classification Using Ensemble Methods*, 2010.
  https://amzn.to/2zxc0F7

A.2 Machine Learning Books

There are a number of books that introduce machine learning and predictive modeling that also cover ensemble learning. These can be helpful but do assume you have some background in probability and linear algebra. Books on machine learning I would recommend include:

- *Data Mining: Practical Machine Learning Tools and Techniques*, 2016.
  https://amzn.to/2YwYsBS
- *Applied Predictive Modeling*, 2013.
  https://amzn.to/2kXE35G
- *The Elements of Statistical Learning*, 2016.
  https://amzn.to/2zAEY72

A.3 Python APIs

It is a great idea to get familiar with the Python APIs that you may use on your ensemble learning projects. Some of the APIs I recommend studying include:

- Scikit-Learn API Reference.
  https://scikit-learn.org/stable/modules/classes.html
- XGBoost API Reference.
  https://xgboost.readthedocs.io/en/latest/python/python_api.html
- LightGBM API Reference.
  https://lightgbm.readthedocs.io/en/latest/Python-API.html

A.4 Ask Questions About Ensemble Learning

What if you need even more help? Below are some resources where you can ask more detailed questions and get helpful technical answers.

- Cross Validated, Machine Learning Questions and Answers.
  https://stats.stackexchange.com/
- Stack Overflow, Programming Questions and Answers.
  https://stackoverflow.com/

A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Write down and think through everything that you have tried and could be related.
- Reduce your question down to the simplest form.
- Search for answers before asking questions.
- Reduce your code down to the smallest possible working example.
- Include working code and error messages in your question.

A.6 Contact the Author

You are not alone. If you ever have any questions about data preparation or the tutorials in this book, please contact me directly. I will do my best to help.

Jason Brownlee
Jason@MachineLearningMastery.com

Appendix B

How to Setup Python on Your Workstation

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

B.1 Tutorial Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
  https://www.continuum.io/
- 2. Click `Anaconda` from the menu and click `Download` to go to the download page.
  https://www.continuum.io/downloads

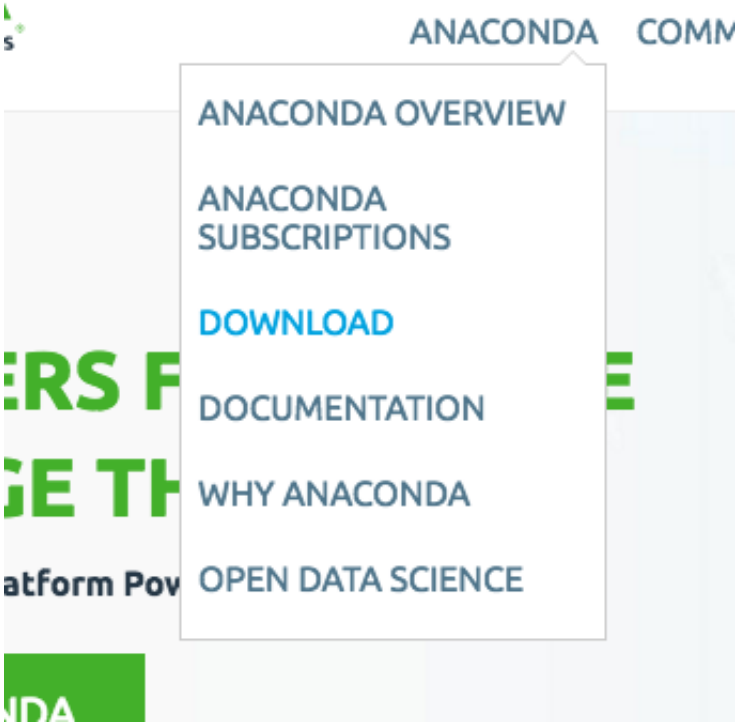

Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
  - Choose Python 3.6
  - Choose the Graphical Installer

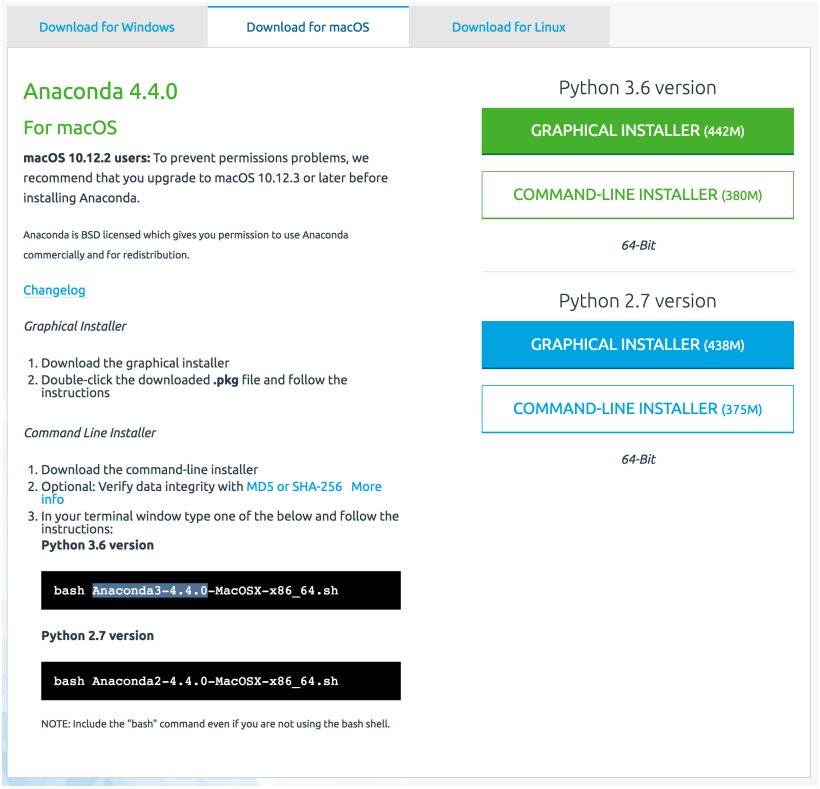

Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on macOS.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

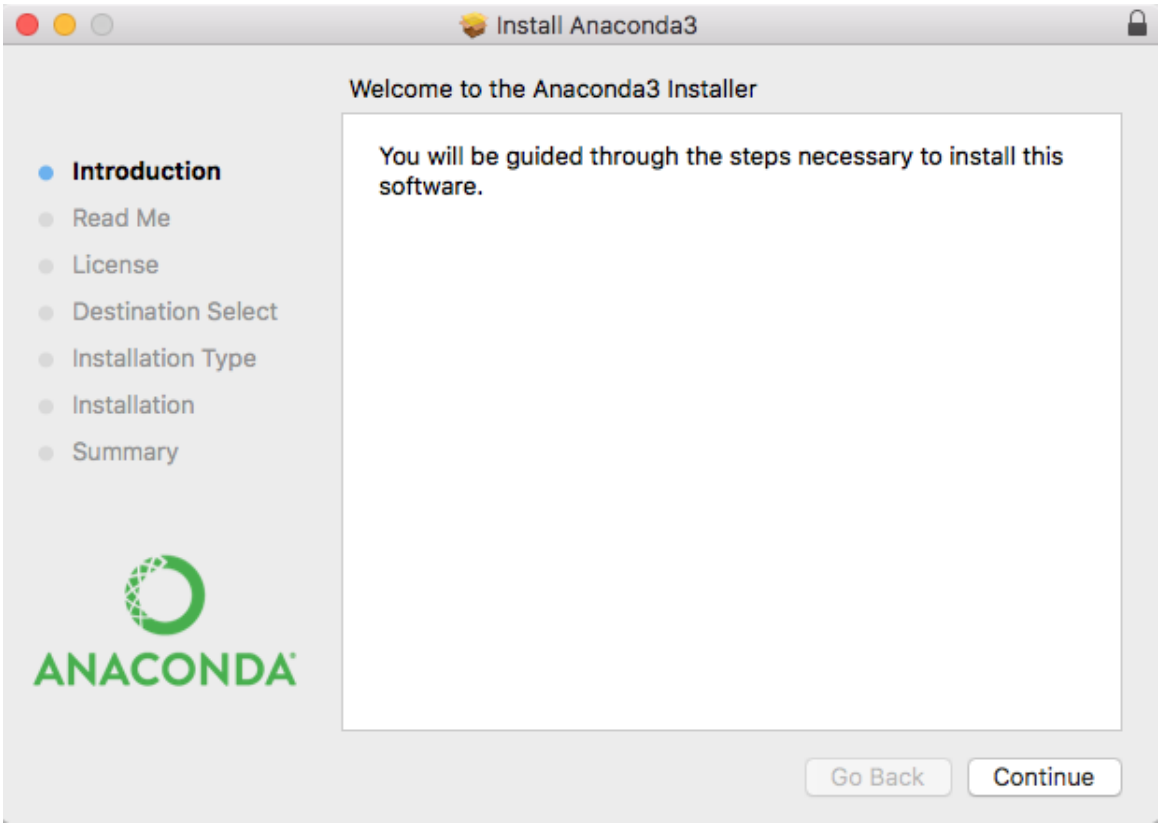

Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

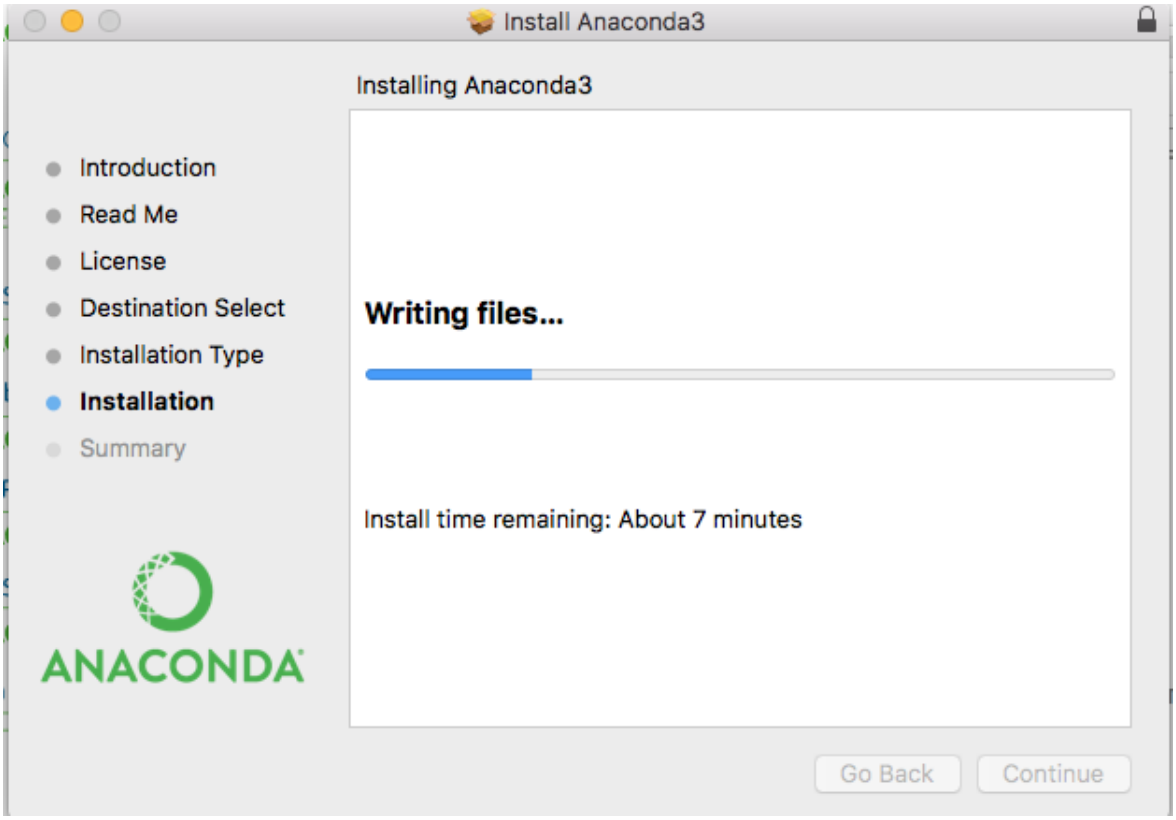

Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

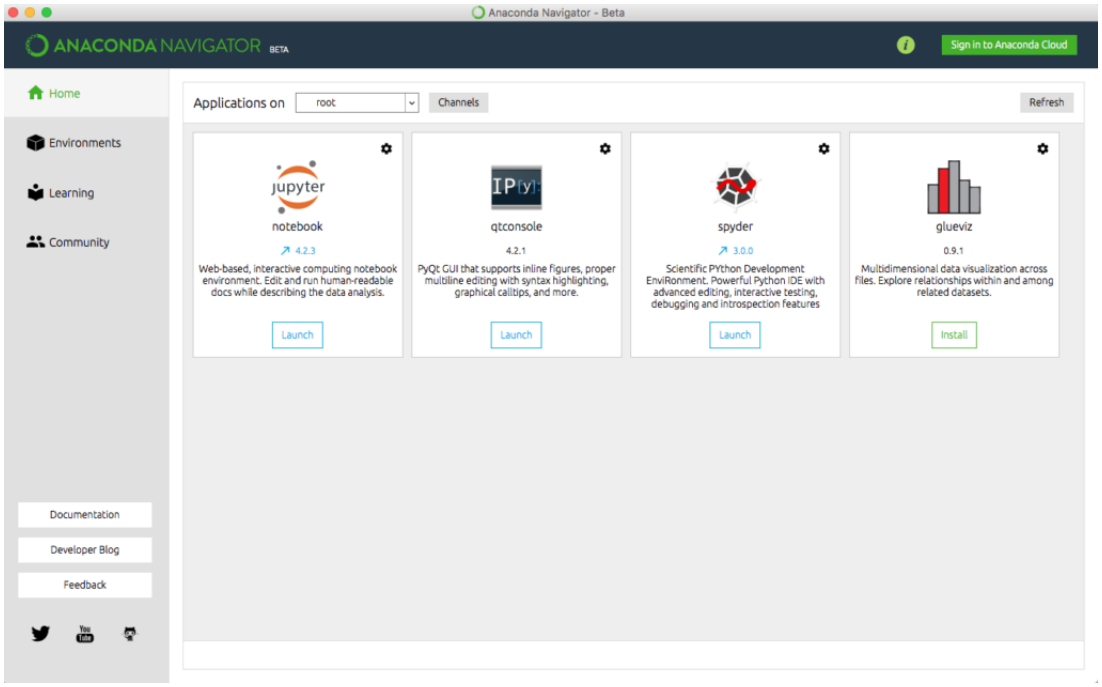

Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -V
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -V
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.4.1
numpy: 1.18.5
matplotlib: 3.3.0
pandas: 1.1.0
statsmodels: 0.11.1
sklearn: 0.23.2
```

Listing B.9: Sample output of versions script.

B.5 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
  https://www.continuum.io/
- Anaconda Navigator.
  https://docs.continuum.io/anaconda/navigator.html
- The conda command line tool.
  http://conda.pydata.org/docs/index.html

B.6 Summary

Congratulations, you now have a working Python development environment for machine learning. You can now learn and practice machine learning on your workstation.

Part IX

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

- The intuition behind drawing upon a crowd or multiple experts when making important decisions and how this intuition carries over to ensemble learning algorithms.
- The benefits of ensemble learning techniques for predictive modeling for both lifting predictive skill and improving model robustness.
- How to develop and evaluate multi-model algorithms for classification and regression problems, providing a precursor to ensemble learning.
- How to develop, configure, and evaluate bagging ensembles for classification and regression predictive modeling problems.
- How to develop and evaluate extensions to bagging, such as random subspace, random forest, and extra trees ensembles.
- How to develop, configure, and evaluate adaptive boosting (AdaBoost) and gradient ensembles for classification and regression predictive modeling problems.
- How to develop and evaluate efficient implementations of gradient boosting ensembles, such as extreme gradient boosting (XGBoost) and light gradient boosting machines (LightGBM).
- How to develop, configure, and evaluate stacking ensembles for classification and regression predictive modeling problems.
- How to develop and evaluate simpler stacking ensembles such as voting and weighted average ensembles.
- How to develop and evaluate extensions to stacking, such as model blending and super learner ensembles.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable foundational skills for ensemble learning. The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey with ensemble learning for machine learning. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2020