

Ensembles: From Beginner to Expert

Russ Conte

2024-06-04

Contents

Chapter 1

Welcome!

Welcome to Ensembles! This book will guide you through the entire process of building your own ensemble models from beginning to end. It will also give you full access to the Ensembles package that automates the entire process.

I've done my very best to make the book very interesting, fun, and practical. There are lots of examples using real world data with all the steps included.

Nature is most accurately
modeled using a diverse set of
individual models and ensembles
of models. This book will give
you tools to make a diverse
range of individual models and
ensembles of models for
numeric, logistic, classification
and time series data.

Figure 1.1: How nature is most accurately modeled

You will be able to do wonderful things as you complete the skills in this book. As the book will show, ensembles are much more accurate than any other method to help us understand and model nature. This will be done with a level of accuracy that has not been achieved previously. And you can do all of it.

The phrase “wonderful things” is very intentional. When Howard Carter was doing archaeology, at one point in November, 1922, he was quite sure he found something important. Carter made a small hole to see through. Lord Carnarvon (who was paying for all of this!) asked Howard Carter, “Can you see anything?”. Howard Carter’s famous reply, “Yes, wonderful things!”. When they opened everything up, they found the intact tomb of Tutankhamun. It contained more than 5,000 items, and enriched our knowledge of ancient Africa beyond any other find.

Here is a tiny taste of one of the more than 5,000 the “wonderful things” found by Howard Carter, Lord Carnarvon, and the team of archaeologists.

I will do my very best to share many “wonderful things” through the entire book as you explore the world of ensembles.

The Ensembles package I’ve made does the entire analysis process for you automatically. This will put the power of ensembles in your hands, give you the strongest foundation for your work, with the highest degree of accuracy.

All of the examples in the book will come from real data. For example (and there are many more examples in the book):

- HR Analytics
- Predicting the winning time in the London Marathon
- World’s most accurate score to a very difficult classification problem
- Beat the best score in student Kaggle competitions

We will have many more practical examples from a very wide range of fields for you to enjoy.

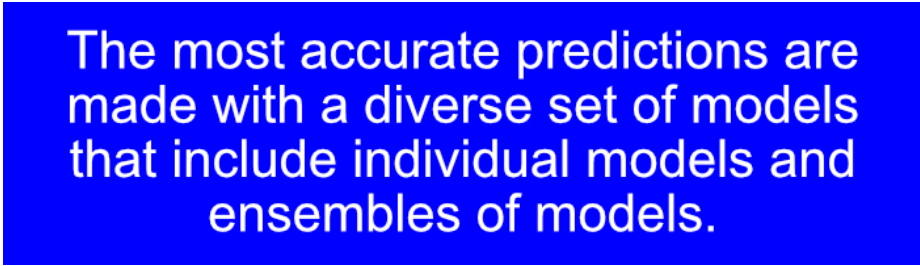
This book will show you how ensembles improve our understanding of nature, and how you can use ensembles in your work. The results using ensembles are much more accurate than has ever been possible before, and that will be demonstrated over and over again in this book. You will be able to use ensembles to understand the world, and build your own models of data, at a level of accuracy that has not been achieved before.

1.1 Ensembles: The New AI, from beginner to expert

As you will see, Ensembles are the new AI. Science has gone from the calculus of Newton and Leibnetz, to differential equations, to the modern world of creating models, and many points in-between. Ensembles are the most powerful way to put models together to achieve the best possible results. This book will guide you through the process, and show you how you can build ensembles that will pass all testing.



Figure 1.2: King Tut Mask



The most accurate predictions are made with a diverse set of models that include individual models and ensembles of models.

Figure 1.3: The most accurate predictions

This is the new AI. Welcome to the path, it's extremely fun, and I look forward to sharing it with you!

1.2 What you will be able to do by the end of the book

- Make your own customized ensembles of models of numerical, classification, logistic and time series data.
- Use the Ensembles package which does the entire process automatically (but little customization is possible).
- Make your ensemble solutions into packages that can be shared with other users.
- Make your ensemble solutions into totally self-contained solutions that can be shared with anyone.
- Learn how ensembles of models can help to make the wisest possible decision based on the data.
- Learn how to present the results at different levels, from a regular user to a CEO and board of directors.
- How to present results that are social media friendly.
- Find your own data and create the ensemble solution from beginning to end (called One Of Your Own in the each of the chapter exercises)
- Solve real world examples in this book where ensembles achieve such results as:
 - Beat the top score in a student data science competition by over 90% (numerical ensembles).
 - Correctly predict the winning time for the 2024 Men's London Marathon (time series ensembles).

1.3. HOW THIS BOOK IS ORGANIZED SO YOU LEARN THE MATERIAL AS EASILY AS POSSIBLE⁹

- Produce a 100% accurate solution to the dry beans classification problem (first in the world with this data set, done using classification ensembles).
- Make recommendations how LeBron James can improve his performance on the basketball court (logistic ensembles).
- Complete a comprehensive Final Project that will put all of your new skills with ensembles together. This result can be shared with employers, advisors, on social media, job interviews, or anywhere else you would like to share your work.

1.3 How this book is organized so you learn the material as easily as possible

The book begins with the foundations of making ensembles of models. We will look at:

- Individual numerical models
- Ensembles of numerical models
- Individual classification models
- Ensembles of classification models
- Individual logistic models
- Ensembles of logistic models
- Individual forecasting models
- Ensembles of forecasting models
- Advanced data visualizations
- Multiple ways to communicate your results. This will range from other people in the field, to customers, to the C-Suite (CEO, CTO, board of directors, etc.)
- We will look at how to treat data science as a business. In particular we will pay close attention to showing return on investment (ROI) in data science, using ensembles of models.
- The book will conclude showing four examples of a final comprehensive project. There will be one example each of numerical data, classification data, logistic data and forecasting data. The example professionally formatted. The source files for each of the eight files are available in a github repository.

1.4 How you can learn the skills as fast as possible: How the exercises are organized

As a young child, I learned that I have much better retention with a system I have always called delayed repetition. This means that I learn best and fastest when I see a worked out example, do several practice examples, and then repeat that after a delay in time. The delay can range from an hour to a few days.

For example, the exercises in the Individual Classification Models chapter will ask you to build models using techniques from the classification models and the prior chapters. The exercises for logistic ensembles will ask you to build models from the content in the logistic models chapter, and each of the previous chapters. It has been my experience that repeating this over and over is the fastest way for me to learn new content, and retain it for the longest period of time.

By the time you get to the Final Comprehensive Project, your skills will be sharp for each of the modeling techniques.

1.5 Going from student to teacher: You are required to post on social media and help others understand the results

One of the most important parts of your role in data science is communicating your findings. I will present many examples of summaries and reports for you to adapt and use on your projects. You are also required to post your results on social media. You may use any appropriate choice of social media, but it needs to be publicly available. This has a number of very important benefits to you:

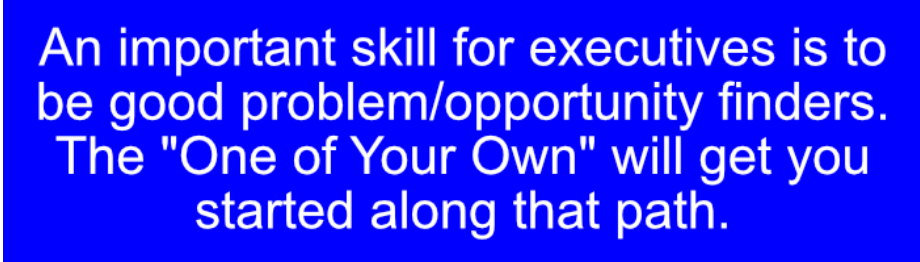
- You will build a body of work that shows your skill level
- The results will demonstrate your ability to communicate in a way that works with a wide variety of people
- You will work to demonstrate very good skills with video and/or audio production
- Use the hashtag #AIEnsembles when you post on social media

1.6 Helping you use the power of pre-trained ensembles and individual models

Another important part of the skills you will learn here includes building pre-trained ensembles and models. The book will walk you through the process of building the pre-trained models and ensembles for each of the four types of data (numerical, classification, logical, and time series).

1.7 Helping you master the material: One of your own exercises

One of the differences with the exercises in Ensembles is the inclusion of One of Your Own exercises. Each set of exercises will include one which asks you to find your own data (with many hints given to help you find data), define the problem, make the ensemble, and report the results.



An important skill for executives is to be good problem/opportunity finders. The "One of Your Own" will get you started along that path.

Figure 1.4: Good problem finder

1.8 Keeping it real: Actual business data and problems as the source of all the data sets

All of the data sets in this book use real data. No exceptions, no synthetic data. The sources of the data are all cited, and the real world implications can be found by a simple search. All of the data is absolutely real.

1.9 Check your biases: Test your model on a neutral data set

Each set of exercises will ask you to check one of the trained models against a neutral data set. If the model has any biases, this should reveal them. Then you'll have the knowledge to go back and address the biases in the models.

1.10 Helping you check your work—and verifying that your results beat previously published results

Many of the data sets have been solved by previous investigators (such as in competitions), so the results here can be easily compared with published results.

For example, we will look at the Boston Housing data set when we look at numerical data sets. This data set has been used many times in Kaggle competitions,

published papers, and Github repositories, among many other sources.

The Ensembles package will automatically solve this data set, and return an RMSE less than 0.20 (there will be slight variation depending on how the parameters are set, as will be explained those chapters). In comparison, the Boston Housing data set was used in this Kaggle student competition: <https://www.kaggle.com/competitions/uou-g03784-2022-spring/leaderboard?tab=public>, and the best score was 2.09684. The Ensembles package will beat the best result in that Kaggle student competition by more than 90%. The Ensembles package only requires one line of code.

1.11 Helping you work as a team with fully reproducible ensembles and individual models

A large part of the skills you will learn include how to make results that are reproducible. This will include:

- Multiple random resamplings of the data
- Learning how to test on totally unseen data for both individual and ensemble models
- How to repeat results (for example, 25 times), and report the accuracy of each resampling

For example, you will make ensembles of models, and then use those trained models to make predictions on totally unseen data.

1.12 The Final Comprehensive Project will put everything together for you

As I was studying data science, one of my professors said that the papers I turned in were “good enough to show to the CEO or Board of Directors” of the Fortune 1000 company he worked for. The chapter on the Final Comprehensive Project will share the highest level of skills in the following:

- Truly understanding the business problem
- Being able to convey the very high value that data science brings to the table
- Being able to back up 100% of your claims with rock solid evidence, facts, and clear reasoning
- How to make a truly professional quality presentation worthy of the C-Suite

I’ve had the incredible pleasure of learning many different skills. A few include being able to play on more than 20 musical instruments, communicate in three

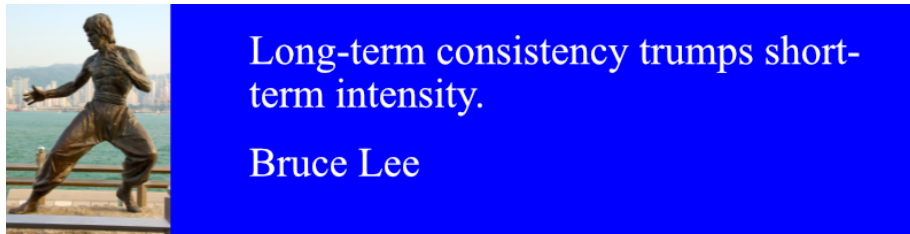


Figure 1.5: Long term consistency wins the race

languages at a professional level, manage a multi-million dollar division of a Fortune 1000 company, run two non-profit volunteer groups, snowboard a three mile run in Colorado, work as a professional counselor, and much more. The book your reading is only my most recent project. None of these skills were acquired overnight. A huge part of the success is being able to make slow and (usually) steady progress. The next chapter will reveal the big secret to getting results, but for now you are best off if you plan some regular time to work on the contents of the book.

Always remember to test everything, that will save you a ton of problems down the road.

1.13 Exercises to help improve your skills

Exercise 1: Schedule regular time to work on this book

You will gain much more progress if you work at a steady pace. Take everything in small pieces. It's OK to go slow, as long as you keep going. Schedule regular time to work on this book, and you will get the largest possible reward for your efforts.

Exercise 2: Read each chapter at least twice **before** you begin working on the material.

Reading each chapter twice before you begin working on it will actually speed up your progress and results. It will actually take less time for you to complete the chapter. You might not believe it right now, but it's totally true.

Exercise 2a: Read a chapter ahead if you are able to do so.

Exercise 3: Read this chapter again

Chapter 2

Introduction and your first ensembles

2.1 How a Chicago blizzard led to the very unlikely story of the best solutions to supervised data

My journey to the most advanced AI in the world started with an actual blizzard in Chicago. It might seem like Chicago would never get a blizzard, but we did in 2011, and it was incredibly intense, as this video shows:

<https://www.youtube.com/watch?v=cPiFn52ztd8>

What does the Chicago 2011 Snomageddon have to do with the creation of the most advanced AI? Everything. Here's the story.

At the time of the 2011 Blizzard I worked a Recruiter for Kelly Services, where I had worked since 1996. I agreed to work out of the Kelly Services office in Frankfort, Illinois at this time, though I worked out of nearly every Kelly Services office at one time or another. The trip to Frankfort involved a daily commute to the office, but I was able to make the best use of the time on the road.

My manager at the time let me know several days in advance that there was a very large amount of snow forecast, and that I might want to be prepared. The most recent forecasts for large amounts of snow in the Chicago area all amounted to nothing. They were perfectly normal days in the Chicago area, so I predicted this storm would also be nothing, based on the most recent results. This was a great example of a prior prediction not transferring well to a current situation.

That morning I went to work as normal, and did not even look at the weather forecast. Around 2:45 pm my manager came out of her office and said “Russ, you need to come here and look at the weather radar!”. I walked into her office, and saw a map of a winter storm that was incredibly huge. She had the image zoomed out, so it was possible to see several states. From what I could tell, the massive snow storm was barreling down on Chicago, and was about 15 minutes away from our location.

I told the candidate I was interviewing that I was leaving immediately, and that he is not allowed to stay. He has to get home as fast as possible for his own safety.

The storm started dropping snow on my trip north back home. The commute took around 50% longer than normal due to the rapidly falling snow.

As I later learned, the storm was forecast to start in the Chicago area around 3:00 pm, finish up between 11:00 am - 1:00 pm two days later, and leave 17 - 19 inches of snow.

How bad was it? Even City of Chicago snow plows were stopped by the snow:



Figure 2.1: Chicago snow plow stuck on Lake Shore Drive in the 2011 snow storm

To see what the forecasts looked like, check out this news report from the day:

2.1. HOW A CHICAGO BLIZZARD LED TO THE VERY UNLIKELY STORY OF THE BEST SOLUTIONS TO SU

<https://www.nbcchicago.com/news/local/blizzard-unleashes-winter-fury/2096753/>

It turns out all three predictions of the blizzard were accurate to a level that almost seemed uncanny to me: Start time, accumulation, and end time were all spot on. This is the first time I recall ever seeing a prediction at this level of accuracy. I had no idea this type of predictive accuracy was even possible. This level of accuracy in predicting results totally blew me away. I had never seen anything with this level of accuracy, and now I wanted to know how it was done.

I searched and searched for how the accuracy was so high for this forecast.

The power of the method—whatever it was—was obvious to me. I realized that if it could work for the weather, the solution method could work in an incredibly broad range of situations. A few of many other areas include business forecasts, production work, modeling prices, and much, much more. But at this point I had no idea how the accurate prediction was done.

Some months later a person wrote to Tom Skilling, chief meteorologist for WGN TV in Chicago. Tom posted an answer that opened up the solution for me. Here is the relevant part of Tom Skilling's answer to a 2011 storm how the forecast was so accurate:

The Weather Service has developed an interesting "SNOWFALL ENSEMBLE FORECAST PROBABILITY SYSTEM" which draws upon a wide range of snow accumulation forecasts from a whole set of different computer models. By "blending" these model projections, probability of snowfalls falling within certain ranges becomes possible. Also, this "blending" of multiple forecasts "smooths" the sometimes huge model disparities in the amounts being predicted. The resulting probabilities therefore represent a "best case" forecast.

So that was the first step. Ensembles were the way they achieved such extraordinary prediction accuracy.

My next goal was to figure out how ensembles were made. As I looked up information, it became obvious that ensembles had been used for a while, such as the winning entry in the Netflix Prize Competition:

The Netflix Prize Competition was sponsored by Netflix to create a method to accurately predict user ratings on films. The minimum winning score needed to beat the Netflix method (named Cinematch) by at least 10%. Several years of work went into solving this problem, and the results even included several published papers. The winning solution was an ensemble of methods that beat the Cinematch results by 10.09%.

So it was now clear to me that ensembles were the path forward. However, I had no idea how to make ensembles.



Figure 2.2: Netflix Prize Competition

I went to graduate school to study data science and predictive analytics. My degree was completed in 2017, from Northwestern University. However, I still was not sure how ensembles of models were built, nor could I find any clear methods to build them (except for pre-made methods, such as random forests). While it is true there were packages that could do some of the work, nothing I found did what I was looking for: How to build ensembles of models in general. Despite playing with the idea and looking online, I was not able to build the ensembles I wanted to build.

2.2 Saturday, October 15, 2022 at 4:58 pm. The exact birth of the Ensembles system

Everything changed on Saturday, October 15, 2022 at 4:58 pm. I was playing with various methods to make an ensemble, and got an ensemble that worked for the very first time. While the results were extremely modest by any standards, it was clear to me that the foundation was there to build a general solution that can work in an extremely wide range of areas. Here is my journal entry:

You might be asking yourself how I know the day and time. That is a very reasonable question. I've been keeping a journal since I was 19 years old, and have thousands of entries. As soon as I realized how to correctly build ensembles, I made this entry, which contains the key elements to make an ensemble, and

It just hit me how to create ensemble models - take all of the y-predicted values from each of the models, make those the X values in a data frame, and have the true y values (in the ensembles data frame) as the true y values, then run that data frame through each of the modeling solutions to find the best performer. W0w!

Figure 2.3: Birth of the ensembles method (typo of W0w in the original)

we will do these steps in just a moment. Notice that the subject line in the journal matches the text above.

One of the ways to improve your skills is to keep a journal, and we'll be looking at that in more depth in this chapter and future chapters. The journal I use is MacJournal, though there are a large number of other options available on the market.

Inspector

Document	Journal	Entry
----------	---------	-------

Topic:

It *just* hit me how to create e

Date:

10/15/2022, 4:58 PM

Tags:

Annotation:

Status:

Unknown

Priority:

None

Due:

☐

5/19/2024, 10:08 AM

Rating:

☒ ⭐⭐⭐⭐⭐

Created:

October 15, 2022, 4:58 PM

Modified:

June 14, 2023, 9:30 PM

Time Edited:

10 minutes 20 seconds

Size:

408 KB

☐ Editable

☐ Flagged

Icon:

Mood:

Word Goal:

Inherit

Label:

Carnation

Background:

Inherit

Blog:

Inherit

Link:

Location:

0.000000

0.000000

Time Zone:

America/Chicago

Related Files:

+ -

Birth of ensembles, Saturday, October 15, 2022 at 4:58 pm

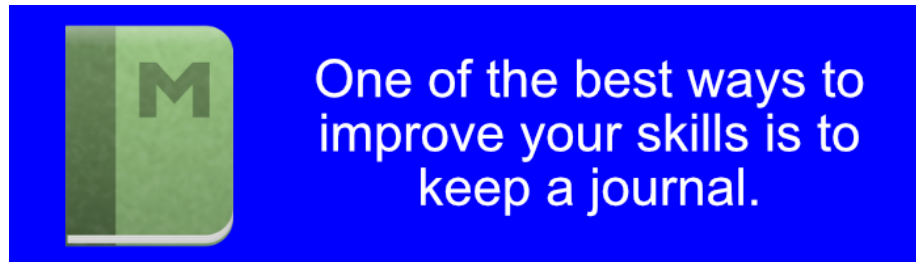


Figure 2.4: Keep a journal

2.3 Here is what an ensemble of models looks like at the most basic level, using the Boston Housing data set as an example:

2.3.1 Head of Boston Housing data set

```
> head(MASS::Boston, n = 10) # Look at the first ten (out of 505) rows of the Boston Housing data set
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2
6	0.02985	0.0	2.18	0	0.458	6.430	58.7	6.0622	3	222	18.7	394.12	5.21	28.7
7	0.08829	12.5	7.87	0	0.524	6.012	66.6	5.5605	5	311	15.2	395.60	12.43	22.9
8	0.14455	12.5	7.87	0	0.524	6.172	96.1	5.9505	5	311	15.2	396.90	19.15	27.1
9	0.21124	12.5	7.87	0	0.524	5.631	100.0	6.0821	5	311	15.2	386.63	29.93	16.5
10	0.17004	12.5	7.87	0	0.524	6.004	85.9	6.5921	5	311	15.2	386.71	17.10	18.9

Figure 2.5: Head of Boston Housing data set

We will start our first ensemble with a data set that only has numerical values. Our first example will use the Boston Housing data set, from the MASS package. While the Boston Housing data set is controversial (and we will discuss some of the controversies in our example making professional quality reports for the C-Suite), for now it works as a very well known data set to begin our journey into ensembles.

Overview of the most basic steps to make an ensemble:

We will be using the Boston Housing data set, so let's have a look at some Boston images:

2.3. HERE IS WHAT AN ENSEMBLE OF MODELS LOOKS LIKE AT THE MOST BASIC LEVEL, USING THE B



Figure 2.6: Boston

2.4 The steps to build your first ensemble from scratch

- Load the packages we will need (MASS, tree)
- Load the Boston Housing data set, and split it into train (60%) and test (40%) sections.
- Create a linear model by fitting the linear model on the training data, and make predictions on the Boston Housing test data. Measure the accuracy of the predictions against the actual values.
- Create a model using trees by fitting the tree model on the training data, and making predictions on the Boston Housing test data. Measure the accuracy of the predictions against the actual values.
- Make a new data frame. This will be our ensemble of model predictions. One column will be the linear predictions, and one will be the tree predictions.
- Make a new column for the true values—these are the true values in the Boston Housing test data set
- Once we have the new ensemble data set, it's simply another data set. No different in many ways from any other data set (except how it was made).
- Break the ensemble data set into train (60%) and test (40%) sections.
- Fit a linear model to the ensemble training data. Make predictions using the testing data, and measure the accuracy of the predictions against the test data.
- Summarize the results.

I suggest reading the over of the most basic steps to make an ensemble a couple of times, to make sure you are very familiar with the steps.

2.5 Building the first actual ensemble

Load the packages we will need (MASS, tree):

```
library(MASS) # for the Boston Housing data set
library(tree) # To make models using trees
library(Metrics) # To calculate error rate (root mean squared error)
library(tidyverse)
#> -- Attaching core tidyverse packages ---- tidyverse 2.0.0 --
#> v dplyr      1.1.4      v readr      2.1.5
#> v forcats    1.0.0      v stringr    1.5.1
#> v ggplot2    3.5.1      v tibble     3.2.1
#> v lubridate  1.9.3      v tidyr      1.3.1
```



```
#> v purrr      1.0.2
#> -- Conflicts ----- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
#> x dplyr::select() masks MASS::select()
#> i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

Load the Boston Housing data set, and split it into train (60%) and test (40%) sections.

```
df <- MASS::Boston
train <- df[1:400, ]
test <- df[401:505, ]

# Let's have a quick look at the train and test sets
head(train)
#>      crim zn indus chas   nox    rm  age    dis rad tax
#> 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900  1 296
#> 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671  2 242
#> 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671  2 242
#> 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622  3 222
#> 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622  3 222
#> 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622  3 222
#>   ptratio  black lstat medv
#> 1    15.3 396.90  4.98 24.0
#> 2    17.8 396.90  9.14 21.6
#> 3    17.8 392.83  4.03 34.7
#> 4    18.7 394.63  2.94 33.4
#> 5    18.7 396.90  5.33 36.2
#> 6    18.7 394.12  5.21 28.7
head(test)
#>      crim zn indus chas   nox    rm  age    dis rad tax
#> 401 25.04610  0 18.1    0 0.693 5.987 100.0 1.5888 24 666
#> 402 14.23620  0 18.1    0 0.693 6.343 100.0 1.5741 24 666
#> 403  9.59571  0 18.1    0 0.693 6.404 100.0 1.6390 24 666
#> 404 24.80170  0 18.1    0 0.693 5.349  96.0 1.7028 24 666
#> 405 41.52920  0 18.1    0 0.693 5.531  85.4 1.6074 24 666
#> 406 67.92080  0 18.1    0 0.693 5.683 100.0 1.4254 24 666
#>   ptratio  black lstat medv
#> 401    20.2 396.90 26.77  5.6
#> 402    20.2 396.90 20.32  7.2
#> 403    20.2 376.11 20.31 12.1
#> 404    20.2 396.90 19.77  8.3
#> 405    20.2 329.46 27.38  8.5
#> 406    20.2 384.97 22.98  5.0
```

Create a linear model by fitting the linear model on the training data, and make predictions on the Boston Housing test data. Measure the accuracy of the predictions against the actual values.

```
Boston_lm <- lm(medv ~ ., data = train) # Fit the model to the training data
Boston_lm_predictions <- predict(object = Boston_lm, newdata = test)

# Let's have a quick look at the model predictions
head(Boston_lm_predictions)
#>      401      402      403      404      405      406
#> 12.618507 19.785728 20.919370 13.014507  6.946392  5.123039
```

Calculate the error for the model

```
Boston_linear_RMSE <- Metrics::rmse(actual = test$medv, predicted = Boston_lm_predictions)
Boston_linear_RMSE
#> [1] 6.108005
```

The error rate for the linear model is 6.108005. Let's do the same using the tree method.

Create a model using trees by fitting the tree model on the training data, and making predictions on the Boston Housing test data. Measure the accuracy of the predictions against the actual values.

```
Boston_tree <- tree(medv ~ ., data = train) # Fit the model to the training data
Boston_tree_predictions <- predict(object = Boston_tree, newdata = test)

# Let's have a quick look at the predictions:
head(Boston_tree_predictions)
#>      401      402      403      404      405      406
#> 13.30769 13.30769 13.30769 13.30769 13.30769 13.30769
```

Calculate the error rate for the tree model:

```
Boston_tree_RMSE <- Metrics::rmse(actual = test$medv, predicted = Boston_tree_predictions)
Boston_tree_RMSE
#> [1] 5.478017
```

The error rate for the tree model is lower (which is better). The error rate for the tree model is 5.478017.

2.6 We're ready to make our first ensemble!!

Make a new data frame. This will be our ensemble of model predictions, and one column for the true values. One column will be the linear predictions, and one will be the tree predictions. We'll make a third column, the true values.

Make a new column for the true values—these are the true values in the Boston

Housing test data set

```
ensemble <- data.frame(
  'linear' = Boston_lm_predictions,
  'tree' = Boston_tree_predictions,
  'y' = test$medv
)

# Let's have a look at the ensemble:
head(ensemble)
#>      linear      tree      y
#> 401 12.618507 13.30769  5.6
#> 402 19.785728 13.30769  7.2
#> 403 20.919370 13.30769 12.1
#> 404 13.014507 13.30769  8.3
#> 405  6.946392 13.30769  8.5
#> 406  5.123039 13.30769  5.0
dim(ensemble)
#> [1] 105  3
```

Once we have the new ensemble data set, it's simply another data set. No different in many ways from any other data set (except how it was made).

Break the ensemble data set into train (60%) and test (40%) sections. There is nothing special about the 60/40 split here, you may use any numbers you wish.

```
ensemble_train <- ensemble[1:60, ]
ensemble_test <- ensemble[61:105, ]

head(ensemble_train)
#>      linear      tree      y
#> 401 12.618507 13.30769  5.6
#> 402 19.785728 13.30769  7.2
#> 403 20.919370 13.30769 12.1
#> 404 13.014507 13.30769  8.3
#> 405  6.946392 13.30769  8.5
#> 406  5.123039 13.30769  5.0
head(ensemble_test)
#>      linear      tree      y
#> 461 23.88984 13.30769 16.4
#> 462 23.29129 13.30769 17.7
#> 463 22.54055 21.84327 19.5
#> 464 25.50940 21.84327 20.2
#> 465 22.71231 21.84327 21.4
#> 466 20.83810 21.84327 19.9
```

Fit a linear model to the ensemble training data. Make predictions using the testing data, and measure the accuracy of the predictions against the test data.

Notice how similar this is to our linear and tree models.

```
# Fit the model to the training data
ensemble_lm <- lm(y ~ ., data = ensemble_train)

# Make predictions using the model on the test data
ensemble_lm_predictions <- predict(object = ensemble_lm, newdata = ensemble_test)

# Calculate error rate for the ensemble predictions
ensemble_lm_rmse <- Metrics::rmse(actual = ensemble_test$y, predicted = ensemble_lm_predictions)

# Report the error rate for the ensemble
ensemble_lm_rmse
#> [1] 4.826962
```

Summarize the results.

```
results <- data.frame(
  'Model' = c('Linear', 'Tree', 'Ensemble'),
  'Error' = c(Boston_linear_RMSE, Boston_tree_RMSE, ensemble_lm_rmse)
)

results
#>      Model      Error
#> 1  Linear 6.108005
#> 2   Tree 5.478017
#> 3 Ensemble 4.826962
```

Clearly the ensemble had the lowest error rate of the three models. The ensemble is easily the best of the three models because it has the lowest error rate of all the models.

2.6.1 Try it yourself: Make an ensemble where the ensemble is made using trees instead of linear models.

```
# Fit the model to the training data
ensemble_tree <- tree(y ~ ., data = ensemble_train)

# Make predictions using the model on the test data
ensemble_tree_predict <- predict(object = ensemble_tree, newdata = ensemble_test)

# Let's look at the predictions
head(ensemble_tree_predict)
#>      461      462      463      464      465      466
#> 14.80000 14.80000 18.94286 18.94286 18.94286 18.94286

# Calculate the error rate
```

2.7. PRINCIPLE: WHAT IS ONE IMPROVEMENT THAT CAN BE MADE? USE A DIVERSE SET OF MODELS

```
ensemble_tree_rmse <- Metrics::rmse(actual = ensemble_test$y, predicted = ensemble_tree_predict)

ensemble_tree_rmse
#> [1] 5.322011
```

How does this compare to our three other results? Let's update the results table

```
results <- data.frame(
  'Model' = c('Linear', 'Tree', 'Ensemble_Linear', 'Ensemble_Tree'),
  'Error' = c(Boston_linear_RMSE, Boston_tree_RMSE, ensemble_lm_rmse, ensemble_tree_rmse)
)

results <- results %>% arrange(Error)

results
#>           Model      Error
#> 1 Ensemble_Linear 4.826962
#> 2 Ensemble_Tree 5.322011
#> 3           Tree 5.478017
#> 4           Linear 6.108005
```

2.6.2 Both of the ensemble models beat both of the individual models in this example

2.7 Principle: What is one improvement that can be made? Use a diverse set of models and ensembles to get the best possible result

As we shall see when we go through and learn how to build ensembles, the numerical method we will use will build 27 individual models and 13 ensembles for a total of 40 results. When the goal is to get the best possible results, a diverse set of models and ensembles, such as the 40 results for numerical data, will produce much better results than a limited number of models and ensembles.

We will do the same principal when we are looking at classification data, logistic, data, and time series forecasting data. We will use a large number of individual models and ensembles with the goal of achieving the best possible result.

2.8 Principle: Randomizing the data before the analysis will make the results more general (and is very easy to do!)

```
df <- df[sample(nrow(df)),] # Randomize the rows before the analysis
```

2.9 Try it yourself: Repeat the previous analysis, but randomize the rows before the analysis. Otherwise keep the process the same. Share your results on social media.

We'll follow the exact same steps, except for randomizing the rows first.

- Randomize the rows
- Break the data into train and test sets
- Fit the model to the training set
- Make predictions and calculate error from the model on the test set

```
df <- df[sample(nrow(df)),] # Randomize the rows before the analysis
```

```
train <- df[1:400, ]
test <- df[401:505, ]
```

```
# Fit the model to the training data
Boston_lm <- lm(medv ~ ., data = train)
```

```
# Make predictions using the model on the test data
Boston_lm_predictions <- predict(object = Boston_lm, newdata = test)
```

```
# Let's have a quick look at the linear model predictions:
```

```
head(Boston_lm_predictions)
#>      191      382      128      18      30      484
#> 30.84277 17.99259 15.32377 16.70613 20.60622 20.58334
```

```
Boston_linear_rmse <- Metrics::rmse(actual = test$medv, predicted = Boston_lm_predictions)
```

```
Boston_tree <- tree(medv ~ ., data = train)
Boston_tree_predictions <- predict(object = Boston_tree, newdata = test)
Boston_tree_rmse <- Metrics::rmse(actual = test$medv, predicted = Boston_tree_predictions)
```

```
# Let's have a quick look at the tree model predictions:
```

2.9. TRY IT YOURSELF: REPEAT THE PREVIOUS ANALYSIS, BUT RANDOMIZE THE ROWS BEFORE THE

```
head(Boston_tree_predictions)
#>      191      382      128      18      30      484
#> 24.26522 11.60615 17.25875 17.25875 20.91807 20.91807

ensemble <- data.frame( 'linear' = Boston_lm_predictions, 'tree' = Boston_tree_predictions, 'y_ensemble' = y_ensemble )

ensemble <- ensemble[sample(nrow(ensemble)), ] # Randomizes the rows of the ensemble

ensemble_train <- ensemble[1:60, ]
ensemble_test <- ensemble[61:105, ]

ensemble_lm <- lm(y_ensemble ~ ., data = ensemble_train)

# Predictions for the ensemble linear model

ensemble_prediction <- predict(ensemble_lm, newdata = ensemble_test)

# Root mean squared error for the ensemble linear model

ensemble_lm_rmse <- Metrics::rmse(actual = ensemble_test$y_ensemble, predicted = ensemble_prediction)

# Same for tree models

ensemble_tree <- tree(y_ensemble ~ ., data = ensemble_train)
ensemble_tree_predictions <- predict(object = ensemble_tree, newdata = ensemble_test)
ensemble_tree_rmse <- Metrics::rmse(actual = ensemble_test$y_ensemble, predicted = ensemble_tree_predictions)

results <- list( 'Linear' = Boston_linear_rmse, 'Trees' = Boston_tree_rmse, 'Ensembles_Linear' = ensemble_lm_rmse, 'Ensembles_Tree' = ensemble_tree_rmse )

results
#> $Linear
#> [1] 4.606489
#>
#> $Trees
#> [1] 4.384072
#>
#> $Ensembles_Linear
#> [1] 3.489345
#>
#> $Ensemble_Tree
#> [1] 5.33012
```

The fact that our results are a bit different from our first ensemble is useful. This gives us another solid principle to use in our analysis methods:

2.10 The more we can randomize the data, the more our results will match nature

Just watch: Repeat the results 100 times, return the mean of the results (hint: It's two small changes)

```
for (i in 1:100) {

  # First the linear model with randomized data

  df <- df[sample(nrow(df)),] # Randomize the rows before the analysis

  train <- df[1:400, ]
  test <- df[401:505, ]

  Boston_lm <- lm(medv ~ ., data = train)
  Boston_lm_predictions <- predict(object = Boston_lm, newdata = test)

  # Let's have a quick look at the linear model predictions:

  head(Boston_lm_predictions)

  # Let's calculate the root mean squared error rate of the predictions:

  Boston_linear_rmse[i] <- Metrics::rmse(actual = test$medv, predicted = Boston_lm_predictions)

  Boston_linear_rmse_mean <- mean(Boston_linear_rmse)

  # Let's use tree models

  Boston_tree <- tree(medv ~ ., data = train)

  Boston_tree_predictions <- predict(object = Boston_tree, newdata = test)

  # Let's have a quick look at the tree model predictions:

  head(Boston_tree_predictions)

  # Let's calculate the root mean squared error rate of the predictions:

  Boston_tree_rmse[i] <- Metrics::rmse(actual = test$medv, predicted = Boston_tree_predictions)
  Boston_tree_rmse_mean <- mean(Boston_tree_rmse)

  ensemble <- data.frame('linear' = Boston_lm_predictions, 'tree' = Boston_tree_predictions)

  ensemble <- ensemble[sample(nrow(ensemble)), ] # Randomizes the rows of the ensemble
```



```

ensemble_train <- ensemble[1:60, ]
ensemble_test  <- ensemble[61:105, ]

# Ensemble linear modeling

ensemble_lm <- lm(y_ensemble ~ ., data = ensemble_train)

# Predictions for the ensemble linear model

ensemble_prediction <- predict(ensemble_lm, newdata = ensemble_test)

# Root mean squared error for the ensemble linear model

ensemble_lm_rmse[i] <- Metrics::rmse(actual = ensemble_test$y_ensemble, predicted = ensemble_prediction)

ensemble_lm_rmse_mean <- mean(ensemble_lm_rmse)

ensemble_tree <- tree(y_ensemble ~ ., data = ensemble_train)

ensemble_tree_predictions <- predict(object = ensemble_tree, newdata = ensemble_test)

ensemble_tree_rmse[i] <- Metrics::rmse(actual = ensemble_test$y_ensemble, predicted = ensemble_tree_predictions)

ensemble_tree_rmse_mean <- mean(ensemble_tree_rmse)

results <- data.frame(
  'Linear' = Boston_linear_rmse_mean,
  'Trees' = Boston_tree_rmse_mean,
  'Ensembles_Linear' = ensemble_lm_rmse_mean,
  'Ensemble_Tree' = ensemble_tree_rmse_mean )
}

results
#>      Linear      Trees Ensembles_Linear Ensemble_Tree
#> 1 4.825244 4.573845      4.186623      5.134664
warnings() # No warnings!

```

2.11 Principle: “Is this my very best work?”

This is your best work to build ensembles at this stage of your skills. We are going to make a number of improvements to the solutions we see here, so our final result will be much stronger than what we have here so far. Always strive

Anything that can be automated, should be
automated. Do as little as possible by hand. Do as
much as possible with functions.
Hadley Wickham and Jenny Bryan in their book, R Packages

Figure 2.7: Automate as much as possible

to do your very best work, without any excuses.

2.12 “Where do I get help with errors or warnings?”

It is extremely useful to check if your code returns any errors or warnings, and fix those as fast as possible. There are numerous sites to help address errors in your code:

<https://stackoverflow.com>

<https://forum.posit.co>

<https://www.r-project.org/help.html>

2.13 Is there an easy way to save all trained models?

Absolutely! We will simply add the code at the end of this section that saves the four trained models (linear, tree, ensemble_linear and ensemble_tree), as follows:

```
library(MASS)
library(Metrics)
library(tree)

ensemble_lm_rmse <- 0
ensemble_tree_rmse <- 0

for (i in 1:100) {

  # Fit the linear model with randomized data

  df <- df[sample(nrow(df)),] # Randomize the rows before the analysis
```

```

train <- df[1:400, ]
test <- df[401:505, ]

Boston_lm <- lm(medv ~ ., data = train)

Boston_lm_predictions <- predict(object = Boston_lm, newdata = test)

# Let's have a quick look at the linear model predictions:

head(Boston_lm_predictions)

# Let's calculate the root mean squared error rate of the predictions:

Boston_linear_rmse[i] <- Metrics::rmse(actual = test$medv, predicted = Boston_lm_predictions)
Boston_linear_rmse_mean <- mean(Boston_linear_rmse)

# Let's use tree models

Boston_tree <- tree(medv ~ ., data = train)

Boston_tree_predictions <- predict(object = Boston_tree, newdata = test)

# Let's have a quick look at the tree model predictions:

head(Boston_tree_predictions)

# Let's calculate the root mean squared error rate of the predictions:

Boston_tree_rmse[i] <- Metrics::rmse(actual = test$medv, predicted = Boston_tree_predictions)
Boston_tree_rmse_mean <- mean(Boston_tree_rmse)

ensemble <- data.frame( 'linear' = Boston_lm_predictions, 'tree' = Boston_tree_predictions, 'y_en

ensemble <- ensemble[sample(nrow(ensemble)), ] # Randomizes the rows of the ensemble

ensemble_train <- ensemble[1:60, ]

ensemble_test <- ensemble[61:105, ]

# Ensemble linear modeling

ensemble_lm <- lm(y_ensemble ~ ., data = ensemble_train)

# Predictions for the ensemble linear model

```

```

ensemble_prediction <- predict(ensemble_lm, newdata = ensemble_test)

# Root mean squared error for the ensemble linear model

ensemble_lm_rmse[i] <- Metrics::rmse(actual = ensemble_test$y_ensemble, predicted = ensemble_prediction)

ensemble_lm_rmse_mean <- mean(ensemble_lm_rmse)

ensemble_tree <- tree(y_ensemble ~ ., data = ensemble_train)

ensemble_tree_predictions <- predict(object = ensemble_tree, newdata = ensemble_test)

ensemble_tree_rmse[i] <- Metrics::rmse(actual = ensemble_test$y_ensemble, predicted = ensemble_tree_predictions)

ensemble_tree_rmse_mean <- mean(ensemble_tree_rmse)

results <- list( 'Linear' = Boston_linear_rmse_mean, 'Trees' = Boston_tree_rmse_mean,
               'Ensembles_Linear' = ensemble_lm_rmse_mean, 'Ensembles_Tree' = ensemble_tree_rmse_mean )

}

results
#> $Linear
#> [1] 4.969499
#>
#> $Trees
#> [1] 4.812232
#>
#> $Ensembles_Linear
#> [1] 4.328306
#>
#> $Ensemble_Tree
#> [1] 5.281115
warnings()

Boston_lm <- Boston_lm
Boston_tree <- Boston_tree
ensemble_lm <- ensemble_lm
ensemble_tree <- ensemble_tree

```

2.13.1 What about classification, logistic and time series data?

In subsequent chapters we will do similar processes with classification, logistic and time series data. It's possible to build ensembles with all these types of data. The results are extremely similar to the results we've seen here with numerical

data: While the ensembles won't always have the best results, it is best to have a diverse set of models and ensembles to get the best possible results.

2.13.2 Principle: Ensembles can work with many types of data, and we will do that in this book

2.13.3 Can it make predictions on totally new data from the trained models—including the ensembles?

The solutions in this book are independent of the use of the data. We will look at everything from housing prices to business analysis to HR analytics to research in medicine. One of our later examples will do exactly what this question is asking—build individual and ensemble models from data, then use those pre-trained models to make predictions on totally unseen data. You will develop this set of skills later in the book, but it's a minor extension of what you're already seen and completed.

2.13.4 The way I was taught how to write code was totally wrong for me: The best way for me is to start at the end and work backward from there. Do not start coding looking for a solution, instead, start with the ending and work backwards from there.

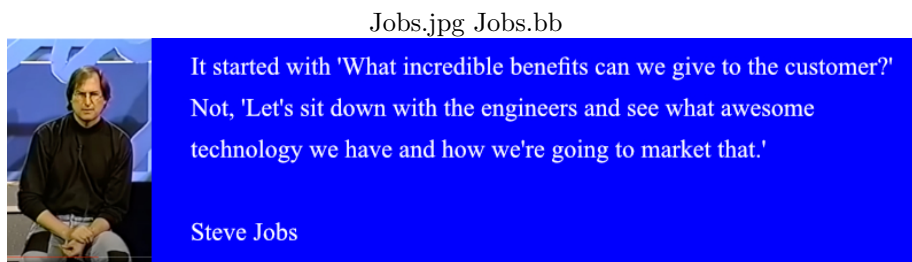


Figure 2.8: Start at the end and work backwards

Start at the end and work backwards from there

The biggest lesson for me in all of this work is how to make ensembles. You've already seen some of the steps, and there are more results to come. The second biggest lesson is that everything I was taught about how to do data science and AI was backwards to what actually works for me in real life. I've learned how I learn, and applied that skill (learning how I learn) to a wide range of skills, including:

- Running a multi-million dollar division of a Fortune 1000 company, including full profit and loss responsibility

- Performing at a professional level on many musical instruments
- Able to communicate in English, Spanish and sign language in a professional setting
- Earning the #1 place on the annual undergraduate university mathematics competition—twice
- Completing a Master’s degree in Guidance and Counseling, allowing me to help many people in their path toward a healthier life
- Leader of the Oak Park, Illinois chapter of Amnesty International for ten years, helping to release several Prisoners of Conscience
- President of the Chicago Apple User Group for ten years, helping many people do extremely good work with their hardware and software
- Leg press 1,000 pounds ten times in a row
- Climbed a mountain in Colorado
- Completed multiple skydives (and looking forward to doing more)

The point here is that I have learned how I learn, and I’ve applied that skill to many areas. When I started learning data science/AI/coding, it was all very different from the way I was being creative my whole life. The way that works for me is to start at the end, work backward from there, and never give up. Maybe the best evidence of the success of this method is this fact:

When I started to write the code that led to the Ensembles package, I followed those steps: Start at the end, work backward from there, and never give up. I wound up writing an average of 1,000 lines of clean, error free code per month for 15 months. The Ensembles package is around 15,000 lines of clean, error free code.

I found my attitude was much more important than my skill set, by a long shot.

2.13.5 How I stuck with it all the way to the end: The best career advice I ever received was from a homeless man I never met, and answers the question of what most strongly predicts success.

Ashford and Simpson

Learning about building ensembles will help you make more accurate predictions. That’s an extrdmely good skill to have in any setting. But I found the most important thing to predict is success. This has been studied, and there are quite a few good works on the subject, both academic and for the general population.

My favorite career advice—which I listened to nearly every day as I worked on the Ensembles project—is from a man who was homeless at the time he came up with the words.

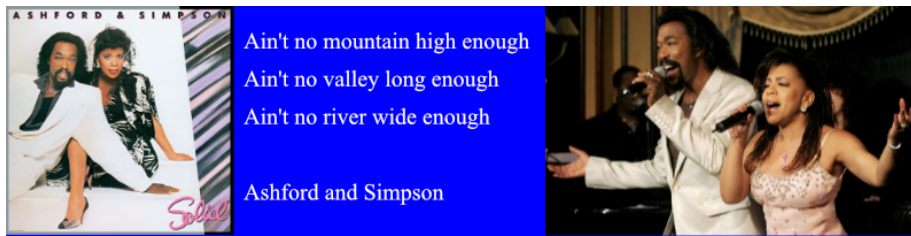


Figure 2.9: Ashford and Simpson

Nick Ashford was from Willow Run, Michigan. He moved to New York, hoping to get into the entertainment world as a dancer. Unfortunately he ended up homeless on the streets of New York. He slept on park benches, and got food from soup kitchens.

He heard that the people at White Rock Baptist Church would feed him (a homeless man) a normal meal, so Nick went there one Sunday morning. He met the people, especially the choir members, and started working with the piano player in the choir. Her name is Valerie Simpson.

Soon Nick and Valerie were writing songs for the church choir. Nick mentioned that while he was homeless, he realized that New York wasn't going to "do me in". He was determined. The words he put down say:

Ain't no mountain high enough

Ain't no valley low enough

Ain't no river wide enough

Valerie took those words, and set them to music. They sent that song to Motown, who released it with Marvin Gaye and Tammy Terrell covering the vocals. It was later re-done by Ashford and Simpson and Paul Riser, with Diana Ross singing the lead.

Here is a short video that summarizes that experience, and concludes with the finale of the 1970 version of the song. This attitude that Ashford and Simpson expressed in song is extremely highly predictive of success, no matter what the field of endeavor. I found this extremely motivating, and used it to overcome any obstacles and challenges I had while on the journey.

While I have the skill of knowing how I learn (which I will continue to share with you in this book), this attitude of working no matter how high the mountain or long the valley or wide the river, gives me the how and the why to keep moving toward success, until that success is fully achieved.

Later on we will look at how to make presentations, consider this as an example of the level of quality that can be done:

<https://www.icloud.com/iclouddrive/002bNfVreagRYCYHAZ9GyQ02w#Ain't%5FNo%5FMountain%5FHigh%5FE>

2.13.6 Exercises:

1. Find your data science Genesis. The data science idea that totally excites you and gets you out of bed every day. The idea that leads to the creation of many other ideas. The biggest and boldest dreams you can possibly have. The idea that is so strong that you have to do it. Not for yourself, but for the benefit of all who will use it and receive all the good it will create.
2. Keep a journal of your progress. It's much easier to see results over time when there is a record. Set the journal up today (or this week). I did not use Github as a journal. My journal was for crazy ideas, contradictory evidence, writing down my frustrations and successes, inspiration, the one next thing I worked on, and having a rock solid record of the path to success. Seeing the path I traversed was a huge motivation to finishing the project.
3. Do your best to add journal entries to your regular schedule.
4. Make an ensemble using the Boston Housing data set. Model any of the other 13 columns of data, not the median value of the home (14th column) which we have been working on in this chapter.
5. Start planning for your comprehensive project. What types of data are you most interested in? What patterns would you like to discover? Begin looking online now for possible data sets, and so a little basic research. More examples will be provided as we get closer to that section of the book.

Chapter 3

Numerical data: How to make 23 individual models, and basic skills with functions

This is where we will begin building the skills to make ensembles of models of numerical data. However, this is going to be much easier than it might appear at first. Let's see how we can make this as easy as possible.

How to work backwards and make the function we need: Start from the end

We are going to start at the ending, not at the beginning, and work backwards from there. This method is much, much easier than working forward, as you will see throughout this book. While it might be a little uncomfortable at first, this skill will allow you to complete your work at a faster rate than if you work forward.

We'll use the Boston Housing data set, and we'll start with the Bagged Random Forest function. For now we're only going to work with one function, to keep everything simple. In essence, we are going to run this like an assembly line.

We want the ending to be the error rate by model. Virtually any customer you work with is going to want to know, "How accurate is it?" That's our starting point.

How do we determine model accuracy? We already did this in the previous chapter, finding the root mean squared error for the individual models and the ensemble models. We're going to do the same steps here, so the process is familiar to you.

To get the error rate by model on the holdout data sets (test and validation), we're going to need a model (Bagged Random Forest in this first example), fit to the training data, and use that model to make predictions on the test data. We can then measure the error in the predictions, just as we did before. These steps should be familiar to you. If not, please re-read the previous chapter.

But what do we need to complete those steps? We're going to have to go backward (a little) and make a function that will allow us to work with any data set.

What does our function need? Let's make a list:

- The data (such as Boston housing)
- Column number (such as 14, the median value of the property)
- Train amount
- Test amount
- Validation amount
- Number of times to resample

One of the key steps here is to change the name of the target variable to `y`. The initial name could be nearly anything, but this method changes the name of the target variable to `y`. This allows us to make one small change that will allow this to be the easiest possible solution:

3.0.1 All our models will be structured the same way: `y ~ ., data = train`

This means that `y` (our target value) is a function of the other features, and the data set is the training data set. While there will be some variations on this in our 27 models, the basic structure is the same.

3.0.2 Having the same structure for all the models makes it much easier to build, debug, and deploy the completed models.

Then we only need to start with our initial values, and it will run.

One extremely nice part about creating models this way is the enormous efficiency it gives us. Once we have the Bagged Random Forest model working, we will be able to use very similar (and identical in many cases!) processes with other models (such as Support Vector Machines).

The rock solid foundation we lay at the beginning will allow us to have a smooth and easy experience once the foundation is solid and we use it to build more models. The other models will mainly be almost exact duplicates of our first example.'

Here are the steps we will follow:

- Load the library
- Set initial values to 0
- Create the function
- Set up random resampling
- Break the data into train and test
- Fit the model on the training data, make predictions and measure error on the test data
- Return the results
- Check for errors or warnings
- Test on a different data set

3.0.3 Exercise: Re-read the steps above how we will work backwards to come up with the function we need.

3.0.4 1. Bagged Random Forest

```
library(e1071) # will allow us to use a tuned random forest model
library(Metrics) # Will allow us to calculate the root mean squared error
library(randomForest) # To use the random forest function
#> randomForest 4.7-1.1
#> Type rfNews() to see new features/changes/bug fixes.
library(tidyverse) # Amazing set of tools for data science
#> -- Attaching core tidyverse packages ---- tidyverse 2.0.0 --
#> v dplyr      1.1.4      v readr      2.1.5
#> v forcats    1.0.0      v stringr   1.5.1
#> v ggplot2    3.5.1      v tibble    3.2.1
#> v lubridate  1.9.3      v tidyr     1.3.1
#> v purrr      1.0.2
#> -- Conflicts ----- tidyverse_conflicts() --
#> x dplyr::combine() masks randomForest::combine()
#> x dplyr::filter()  masks stats::filter()
#> x dplyr::lag()     masks stats::lag()
#> x ggplot2::margin() masks randomForest::margin()
#> i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
# Set initial values to 0. The function will return an error if any of these are left out.

bag_rf_holdout_RMSE <- 0
bag_rf_holdout_RMSE_mean <- 0
bag_rf_train_RMSE <- 0
```

```

bag_rf_test_RMSE <- 0
bag_rf_validation_RMSE <- 0

# Define the function

numerical_1 <- function(data, colnum, train_amount, test_amount, numresamples){

#Set up random resampling

for (i in 1:numresamples) {

# Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col())
df <- df[sample(nrow(df)), ] # randomizes the rows

#Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
train <- df[idx == 1, ]
test <- df[idx == 2, ]

# Fit the model to the training data, make predictions on the testing data, then calcu
bag_rf_train_fit <- e1071::tune.randomForest(x = train, y = train$y, mtry = ncol(train
bag_rf_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict( object =
bag_rf_train_RMSE_mean <- mean(bag_rf_train_RMSE)
bag_rf_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict( object = bag
bag_rf_test_RMSE_mean <- mean(bag_rf_test_RMSE)

# Itemize the error on the holdout data sets, and calculate the mean of the results
bag_rf_holdout_RMSE[i] <- mean(bag_rf_test_RMSE_mean)
bag_rf_holdout_RMSE_mean <- mean(c(bag_rf_holdout_RMSE))

# These are the predictions we will need when we make the ensembles
bag_rf_test_predict_value <- as.numeric(predict(object = bag_rf_train_fit$best.model,

# Return the mean of the results to the user

} # closing brace for numresamples
  return(bag_rf_holdout_RMSE_mean)

} # closing brace for numerical_1 function

```

```
# Here is our first numerical function in actual use. We will use 25 resamples
numerical_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 25)
#> [1] 0.3032684
warnings() # no warnings, the best possible result
```

Exercise: Try it yourself: Change the values of train, test and validation, and the number of resamples. See how those change the result.

One of your own: Find any numerical data set, and make a bagged random forest function for that data set. (For example, you may use the Auto data set in the ISLR package. You will need to remove the last column, vehicle name. Model mpg as a function of the other features using the Bagged Random Forest function, but any numerical data set will work).

Post: Share on social your first results making a numerical function (screen shot/video optional at this stage, we will be learning how to do those later)

For example, “Did my first data science function building up to making ensembles later on. Got everything to run, no errors. #AIEnsembles”

Now we will build the remaining 22 models for numerical data. They are all built using the same structure, on the same foundation.

Now that we know how to build a basic function, let’s build the 22 other sets of tools we will need to make our ensemble, starting with bagging:

3.0.5 2. Bagging (bootstrap aggregating)

```
library(ipred) #for the bagging function

# Set initial values to 0
bagging_train_RMSE <- 0
bagging_test_RMSE <- 0
bagging_validation_RMSE <- 0
bagging_holdout_RMSE <- 0
bagging_test_predict_value <- 0
bagging_validation_predict_value <- 0

#Create the function:

bagging_1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

#Set up random resampling
for (i in 1:numresamples) {

#Changes the name of the target column to y
```

```

y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the th

# Breaks the data into train and test sets

idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
train <- df[idx == 1, ]
test <- df[idx == 2, ]

# Fit the model to the training data, calculate error, make predictions on the holdout

bagging_train_fit <- ipred::bagging(formula = y ~ ., data = train)
bagging_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = 1
bagging_train_RMSE_mean <- mean(bagging_train_RMSE)
bagging_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = ba
bagging_test_RMSE_mean <- mean(bagging_test_RMSE)
bagging_holdout_RMSE[i] <- mean(bagging_test_RMSE_mean)
bagging_holdout_RMSE_mean <- mean(bagging_holdout_RMSE)
y_hat_bagging <- c(bagging_test_predict_value)

} # closing braces for the resampling function
  return(bagging_holdout_RMSE_mean)

} # closing braces for the bagging function

# Test the function:
bagging_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.20, n
#> [1] 4.269031
warnings() # no warnings

```

3.0.6 3. BayesGLM

```

library(arm) # to use bayesglm function
#> Loading required package: MASS
#>
#> Attaching package: 'MASS'
#> The following object is masked from 'package:dplyr':
#>
#>     select
#> Loading required package: Matrix
#>

```

```

#> Attaching package: 'Matrix'
#> The following objects are masked from 'package:tidyr':
#>
#>     expand, pack, unpack
#> Loading required package: lme4
#>
#> arm (Version 1.14-4, built: 2024-4-1)
#> Working directory is /Users/russconte/Library/Mobile Documents/com~apple~CloudDocs/Documents/1

# Set initial values to 0
bayesglm_train_RMSE <- 0
bayesglm_test_RMSE <- 0
bayesglm_validation_RMSE <- 0
bayesglm_holdout_RMSE <- 0
bayesglm_test_predict_value <- 0
bayesglm_validation_predict_value <- 0

# Create the function:
bayesglm_1 <- function(data, colnum, train_amount, test_amount, numresamples){

#Set up random resampling
for (i in 1:numresamples) {

#Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

#Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column

#Breaks the data into train, test and validation sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
train <- df[idx == 1, ]
test <- df[idx == 2, ]

bayesglm_train_fit <- arm::bayesglm(y ~ ., data = train, family = gaussian(link = "identity"))
bayesglm_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = bayesglm_train_fit))
bayesglm_train_RMSE_mean <- mean(bayesglm_train_RMSE)
bayesglm_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = bayesglm_train_fit))
bayesglm_test_RMSE_mean <- mean(bayesglm_test_RMSE)
y_hat_bayesglm <- c(bayesglm_test_predict_value)

} # closing braces for resampling
return(bayesglm_test_RMSE_mean)

```

```

} # closing braces for the function

bayesglm_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.20, n
#> [1] 4.885836
warnings() # no warnings

```

3.0.7 4. BayesRNN

```

library(brnn) # so we can use the BayesRNN function
#> Loading required package: Formula
#> Loading required package: truncnorm

#Set initial values to 0

bayesrnn_train_RMSE <- 0
bayesrnn_test_RMSE <- 0
bayesrnn_validation_RMSE <- 0
bayesrnn_holdout_RMSE <- 0
bayesrnn_test_predict_value <- 0
bayesrnn_validation_predict_value <- 0

# Create the function:

bayesrnn_1 <- function(data, colnum, train_amount, test_amount, numresamples){

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the
df <- df[sample(nrow(df)), ] # randomizes the rows

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
train <- df[idx == 1, ]
test <- df[idx == 2, ]

# Fit the model on the training data, make predictions on the testing data
bayesrnn_train_fit <- brnn::brnn(x = as.matrix(train), y = train$y)
bayesrnn_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object =

```



```

bayesrnn_train_RMSE_mean <- mean(bayesrnn_train_RMSE)
bayesrnn_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = bayesrnn_train, newdata = test))
bayesrnn_test_RMSE_mean <- mean(bayesrnn_test_RMSE)

y_hat_bayesrnn <- c(bayesrnn_test_predict_value)

} # Closing brace for number of resamples
  return(bayesrnn_test_RMSE_mean)

} # Closing brace for the function

bayesrnn_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 100)
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015619
#> gamma= 31.4899   alpha= 4.1846   beta= 14577.8
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015038
#> gamma= 30.848   alpha= 4.5454   beta= 26230.68
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.701735
#> gamma= 30.5029   alpha= 3.595   beta= 13213.31
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015669
#> gamma= 31.482   alpha= 4.8155   beta= 15715.61
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016138
#> gamma= 31.5555   alpha= 4.7175   beta= 18489.23
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016301
#> gamma= 31.2605   alpha= 5.5195   beta= 17152.98
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015371
#> gamma= 31.2354   alpha= 3.859   beta= 17266.54
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015323
#> gamma= 30.6929   alpha= 4.5552   beta= 14882.59
#> Number of parameters (weights and biases) to estimate: 32

```

```

#> Nguyen-Widrow method
#> Scaling factor= 0.7016085
#> gamma= 31.1701    alpha= 5.6211    beta= 13769.31
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016926
#> gamma= 30.7758    alpha= 4.7287    beta= 14626.84
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015619
#> gamma= 31.0773    alpha= 5.2727    beta= 14593.56
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015979
#> gamma= 31.1685    alpha= 4.867    beta= 21284.21
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015179
#> gamma= 31.2714    alpha= 5.3142    beta= 18404.69
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016636
#> gamma= 30.733    alpha= 4.0453    beta= 54421.22
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015979
#> gamma= 31.0064    alpha= 4.8968    beta= 15219.09
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016523
#> gamma= 30.6649    alpha= 4.0476    beta= 36779.5
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016467
#> gamma= 31.6033    alpha= 5.099    beta= 13526.3
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016579
#> gamma= 31.4734    alpha= 4.6877    beta= 15333.88
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016986
#> gamma= 30.0645    alpha= 5.5005    beta= 13039.4
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method

```

```

#> Scaling factor= 0.7016411
#> gamma= 31.3163    alpha= 5.5848    beta= 14275.6
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016246
#> gamma= 31.2516    alpha= 5.4363    beta= 15867.21
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015771
#> gamma= 30.1966    alpha= 4.2673    beta= 21632.2
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015323
#> gamma= 31.1172    alpha= 5.0178    beta= 14779.09
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7016246
#> gamma= 31.5115    alpha= 5.8685    beta= 14682.03
#> Number of parameters (weights and biases) to estimate: 32
#> Nguyen-Widrow method
#> Scaling factor= 0.7015371
#> gamma= 31.137     alpha= 4.6982    beta= 17921.46
#> [1] 0.1486326

warnings() # no warnings for BayesRNN function

```

3.0.8 5. Boosted Random Forest

```

library(e1071)
library(randomForest)
library(tidyverse)

#Set initial values to 0
boost_rf_train_RMSE <- 0
boost_rf_test_RMSE <- 0
boost_rf_validation_RMSE <- 0
boost_rf_holdout_RMSE <- 0
boost_rf_test_predict_value <- 0
boost_rf_validation_predict_value <- 0

#Create the function:
boost_rf_1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

#Set up random resampling

```

```

for (i in 1:numresamples) {

  #Changes the name of the target column to y
  y <- 0
  colnames(data)[colnum] <- "y"

  #Moves the target column to the last column on the right
  df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the
  df <- df[sample(nrow(df)), ] # randomizes the rows

  # Breaks the data into train and test sets
  idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
  train <- df[idx == 1, ]
  test <- df[idx == 2, ]

  # Fit boosted random forest model on the training data, make predictions on holdout da

  boost_rf_train_fit <- e1071::tune.randomForest(x = train, y = train$y, mtry = ncol(trai
  boost_rf_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict( object =
    ))
  boost_rf_train_RMSE_mean <- mean(boost_rf_train_RMSE)
  boost_rf_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict( object =
    ))
  boost_rf_test_RMSE_mean <- mean(boost_rf_test_RMSE)

} # closing brace for numresamples
  return(boost_rf_test_RMSE_mean)

} # closing brace for the function

boost_rf_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, n
#> [1] 0.3284326
warnings() # no warnings for Boosted Random Forest function

```

3.0.9 6. Cubist

```

library(Cubist)
#> Loading required package: lattice
library(tidyverse)

# Set initial values to 0

cubist_train_RMSE <- 0
cubist_test_RMSE <- 0

```

```

cubist_validation_RMSE <- 0
cubist_holdout_RMSE <- 0
cubist_test_predict_value <- 0

# Create the function:

cubist_1 <- function(data, colnum, train_amount, test_amount, numresamples){

  #Set up random resampling
  for (i in 1:numresamples) {

    # Changes the name of the target column to y
    y <- 0
    colnames(data)[colnum] <- "y"

    # Moves the target column to the last column on the right
    df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column
    df <- df[sample(nrow(df)), ] # randomizes the rows

    # Breaks the data into train and test sets
    idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
    train <- df[idx == 1, ]
    test <- df[idx == 2, ]

    # Fit the model on the training data, make predictions on the holdout data
    cubist_train_fit <- Cubist::cubist(x = train[, 1:ncol(train) - 1], y = train$y)
    cubist_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = cubist_train_fit))
    cubist_train_RMSE_mean <- mean(cubist_train_RMSE)
    cubist_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = cubist_train_fit))
    cubist_test_RMSE_mean <- mean(cubist_test_RMSE)

  } # closing braces for numresamples
  return(cubist_test_RMSE_mean)

} # closing braces for the function

cubist_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 1000)
#> [1] 4.57873
warnings() # no warnings for individual cubist function

```

3.0.10 7. Elastic

```

library(glmnet) # So we can run the elastic model
#> Loaded glmnet 4.1-8

```

```

library(tidyverse)

# Set initial values to 0

elastic_train_RMSE <- 0
elastic_test_RMSE <- 0
elastic_validation_RMSE <- 0
elastic_holdout_RMSE <- 0
elastic_test_predict_value <- 0
elastic_validation_predict_value <- 0
elastic_test_RMSE <- 0
elastic_test_RMSE_df <- data.frame(elastic_test_RMSE)
elastic_validation_RMSE <- 0
elastic_validation_RMSE_df <- data.frame(elastic_validation_RMSE)
elastic_holdout_RMSE <- 0
elastic_holdout_RMSE_df <- data.frame(elastic_holdout_RMSE)

# Create the function:
elastic_1 <- function(data, colnum, train_amount, test_amount, validation_amount, numr

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the

# Breaks the data into train, test and validation sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
train <- df[idx == 1,]
test <- df[idx == 2,]

# Set up the elastic model

y <- train$y
x <- data.matrix(train %>% dplyr::select(-y))
elastic_model <- glmnet::glmnet(x, y, alpha = 0.5)
elastic_cv <- cv.glmnet(x, y, alpha = 0.5)
best_elastic_lambda <- elastic_cv$lambda.min
best_elastic_model <- glmnet::glmnet(x, y, alpha = 0, lambda = best_elastic_lambda)
elastic_test_pred <- predict(best_elastic_model, s = best_elastic_lambda, newx = data.

```

```

elastic_test_RMSE <- Metrics::rmse(actual = test$y, predicted = elastic_test_pred)
elastic_test_RMSE_df <- rbind(elastic_test_RMSE_df, elastic_test_RMSE)
elastic_test_RMSE_mean <- mean(elastic_test_RMSE_df$elastic_test_RMSE[2:nrow(elastic_test_RMSE_df)])

elastic_holdout_RMSE <- mean(elastic_test_RMSE_mean)
elastic_holdout_RMSE_df <- rbind(elastic_holdout_RMSE_df, elastic_holdout_RMSE)
elastic_holdout_RMSE_mean <- mean(elastic_holdout_RMSE_df$elastic_holdout_RMSE[2:nrow(elastic_holdout_RMSE_df)])

} # closing brace for numresample
  return(elastic_holdout_RMSE_mean)

} # closing brace for the elastic function

elastic_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 100)
#> [1] 4.972343
warnings() # no warnings for individual elastic function

```

3.0.11 8. Generalized Additive Models with smoothing splines

```

library(gam) # for fitting generalized additive models
#> Loading required package: splines
#> Loading required package: foreach
#>
#> Attaching package: 'foreach'
#> The following objects are masked from 'package:purrr':
#>
#>   accumulate, when
#> Loaded gam 1.22-3

# Set initial values to 0

gam_train_RMSE <- 0
gam_test_RMSE <- 0
gam_holdout_RMSE <- 0
gam_test_predict_value <- 0

# Create the function:
gam1 <- function(data, colnum, train_amount, test_amount, numresamples){

# Set up random resampling

for (i in 1:numresamples) {

```

```

# Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right

df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
train <- df[idx == 1,]
test <- df[idx == 2, ]

# Set up to fit the model on the training data

n_unique_vals <- purrr::map_dbl(df, dplyr::n_distinct)

# Names of columns with >= 4 unique vals
keep <- names(n_unique_vals)[n_unique_vals >= 4]

gam_data <- df %>% dplyr::select(dplyr::all_of(keep))

# Model data

train1 <- train %>% dplyr::select(dplyr::all_of(keep))

test1 <- test %>% dplyr::select(dplyr::all_of(keep))

names_df <- names(gam_data[, 1:ncol(gam_data) - 1])
f2 <- stats::as.formula(paste0("y ~", paste0("gam::s(", names_df, ")", collapse = "+"))

gam_train_fit <- gam::gam(f2, data = train1)
gam_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = gam_t
gam_train_RMSE_mean <- mean(gam_train_RMSE)
gam_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = gam_tr
gam_test_RMSE_mean <- mean(gam_test_RMSE)
gam_holdout_RMSE[i] <- mean(gam_test_RMSE_mean)
gam_holdout_RMSE_mean <- mean(gam_holdout_RMSE)

} # closing braces for numresamples
  return(gam_holdout_RMSE_mean)

} # closing braces for gam function

gam1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numres

```



```
#> [1] 5.077224
warnings() # no warnings for individual gam function
```

3.0.12 9. Gradient Boosted

```
library(gbm) # to allow use of gradient boosted models
#> Loaded gbm 2.1.9
#> This version of gbm is no longer under development. Consider transitioning to gbm3, https://github.com/gbm/gbm3

# Set initial values to 0
gb_train_RMSE <- 0
gb_test_RMSE <- 0
gb_validation_RMSE <- 0
gb_holdout_RMSE <- 0
gb_test_predict_value <- 0
gb_validation_predict_value <- 0

gb1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

  # Set up random resampling
  for (i in 1:numresamples) {

    # Changes the name of the target column to y
    y <- 0
    colnames(data)[colnum] <- "y"

    # Moves the target column to the last column on the right
    df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column

    # Breaks the data into train and test sets
    idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
    train <- df[idx == 1,]
    test <- df[idx == 2,]

    gb_train_fit <- gbm::gbm(train$y ~ ., data = train, distribution = "gaussian", n.trees = 100, shrinkage = 0.1)
    gb_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = gb_train_fit, newdata = train))
    gb_train_RMSE_mean <- mean(gb_train_RMSE)
    gb_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = gb_train_fit, newdata = test))
    gb_test_RMSE_mean <- mean(gb_test_RMSE)

  } # closing brace for numresamples
  return(gb_test_RMSE_mean)

} # closing brace for gb1 function
```

```
gbl(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresam
```

[illegible]

```
#> Using 100 trees...
#>
#> Using 100 trees...
#>
#> Using 100 trees...
#>
#> Using 100 trees...
#>
#> Using 100 trees...
#> [1] 3.383987
warnings() # no warnings for individual gradient boosted function
```

3.0.13 10. K-Nearest Neighbors (tuned)

```
library(e1071)

# Set initial values to 0
knn_train_RMSE <- 0
knn_test_RMSE <- 0
knn_validation_RMSE <- 0
knn_holdout_RMSE <- 0
knn_test_predict_value <- 0
knn_validation_predict_value <- 0

knn1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamp) {

  # Set up random resampling
  for (i in 1:numresamples) {

    # Changes the name of the target column to y

    y <- 0
    colnames(data)[colnum] <- "y"

    # Moves the target column to the last column on the right

    df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the

    # Breaks the data into train and test sets
    idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
    train <- df[idx == 1,]
    test <- df[idx == 2,]

    knn_train_fit <- e1071::tune.gknn(x = train[, 1:ncol(train) - 1], y = train$y, scale =
```

```

knn_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict( object = knn_train_fit$
  newdata = train[, 1:ncol(train) - 1], k = knn_train_fit$best_model$k))
knn_train_RMSE_mean <- mean(knn_train_RMSE)
knn_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict( object = knn_train_fit$be
  k = knn_train_fit$best_model$k, newdata = test[, 1:ncol(test) - 1]))
knn_test_RMSE_mean <- mean(knn_test_RMSE)
knn_holdout_RMSE[i] <- mean(c(knn_test_RMSE_mean))
knn_holdout_RMSE_mean <- mean(knn_holdout_RMSE)

} # closing brace for numresamples
  return(knn_holdout_RMSE_mean)

} # closing brace for knn1 function

knn1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 25)
#> [1] 6.724675
warnings() # no warnings for individual knn function

```

3.0.14 11. Lasso

```

library(glmnet) # So we can run the lasso model

# Set initial values to 0

lasso_train_RMSE <- 0
lasso_test_RMSE <- 0
lasso_validation_RMSE <- 0
lasso_holdout_RMSE <- 0
lasso_test_predict_value <- 0
lasso_validation_predict_value <- 0
lasso_test_RMSE <- 0
lasso_test_RMSE_df <- data.frame(lasso_test_RMSE)
lasso_validation_RMSE <- 0
lasso_validation_RMSE_df <- data.frame(lasso_validation_RMSE)
lasso_holdout_RMSE <- 0
lasso_holdout_RMSE_df <- data.frame(lasso_holdout_RMSE)

# Create the function:
lasso_1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y

```

```

y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the
df <- df[sample(nrow(df)), ] # randomizes the rows

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
train <- df[idx == 1, ]
test <- df[idx == 2, ]

# Set up the lasso model
y <- train$y
x <- data.matrix(train %>% dplyr::select(-y))
lasso_model <- glmnet::glmnet(x, y, alpha = 1.0)
lasso_cv <- cv.glmnet(x, y, alpha = 1.0)
best_lasso_lambda <- lasso_cv$lambda.min
best_lasso_model <- glmnet::glmnet(x, y, alpha = 0, lambda = best_lasso_lambda)
lasso_test_pred <- predict(best_lasso_model, s = best_lasso_lambda, newx = data.matrix

lasso_test_RMSE <- Metrics::rmse(actual = test$y, predicted = lasso_test_pred)
lasso_test_RMSE_df <- rbind(lasso_test_RMSE_df, lasso_test_RMSE)
lasso_test_RMSE_mean <- mean(lasso_test_RMSE_df$lasso_test_RMSE[2:nrow(lasso_test_RMSE

lasso_holdout_RMSE <- mean(lasso_test_RMSE_mean)
lasso_holdout_RMSE_df <- rbind(lasso_holdout_RMSE_df, lasso_holdout_RMSE)
lasso_holdout_RMSE_mean <- mean(lasso_holdout_RMSE_df$lasso_holdout_RMSE[2:nrow(lasso_I

} # closing brace for numresample
  return(lasso_holdout_RMSE_mean)

} # closing brace for the lasso_1 function

lasso_1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, num
#> [1] 4.830226
warnings() # no warnings for individual lasso function

```

3.0.15 12. Linear (tuned)

```

library(e1071) # for tuned linear models

# Set initial values to 0
linear_train_RMSE <- 0

```

```

linear_test_RMSE <- 0
linear_holdout_RMSE <- 0

# Set up the function
linear1 <- function(data, colnum, train_amount, test_amount, numresamples){

# Set up random resampling
for (i in 1:numresamples) {

#Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column
df <- df[sample(nrow(df)), ] # randomizes the rows

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
train <- df[idx == 1,]
test <- df[idx == 2, ]

linear_train_fit <- e1071::tune.rpart(formula = y ~ ., data = train)
linear_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = linear_train_fit, newdata = train))
linear_train_RMSE_mean <- mean(linear_train_RMSE)
linear_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = linear_train_fit, newdata = test))
linear_holdout_RMSE_mean <- mean(linear_test_RMSE)

} # closing brace for numresamples
return(linear_holdout_RMSE_mean)

} # closing brace for linear1 function

linear1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 1000)
#> [1] 4.879767
warnings() # no warnings for individual lasso function

```

3.0.16 13. LQS

```

library(MASS) # to allow us to run LQS models

# Set initial values to 0

lqs_train_RMSE <- 0

```

```

lqs_test_RMSE <- 0
lqs_validation_RMSE <- 0
lqs_holdout_RMSE <- 0
lqs_test_predict_value <- 0
lqs_validation_predict_value <- 0

lqs1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamp)

  # Set up random resampling
  for (i in 1:numresamples) {

    # Changes the name of the target column to y
    y <- 0
    colnames(data)[colnum] <- "y"

    # Moves the target column to the last column on the right
    df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the

    # Breaks the data into train and test sets
    idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
    train <- df[idx == 1,]
    test <- df[idx == 2, ]

    lqs_train_fit <- MASS::lqs(train$y ~ ., data = train)
    lqs_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = lqs_
    lqs_train_RMSE_mean <- mean(lqs_train_RMSE)
    lqs_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = lqs_tr
    lqs_test_RMSE_mean <- mean(lqs_test_RMSE)

    y_hat_lqs <- c(lqs_test_predict_value, lqs_validation_predict_value)

  } # Closing brace for numresamples
    return(lqs_test_RMSE_mean)

} # Closing brace for lqs1 function

lqs1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numres
#> [1] 7.424303
warnings() # no warnings for individual lqs function

```

3.0.17 14. Neuralnet

```

library(neuralnet)
#>

```



```

#> Attaching package: 'neuralnet'
#> The following object is masked from 'package:dplyr':
#>
#>      compute

#Set initial values to 0

neuralnet_train_RMSE <- 0
neuralnet_test_RMSE <- 0
neuralnet_validation_RMSE <- 0
neuralnet_holdout_RMSE <- 0
neuralnet_test_predict_value <- 0
neuralnet_validation_predict_value <- 0

# Fit the model to the training data
neuralnet1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y

y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column

# Breaks the data into train and test data sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
train <- df[idx == 1,]
test <- df[idx == 2, ]

maxs <- apply(df, 2, max)
mins <- apply(df, 2, min)
scaled <- as.data.frame(scale(df, center = mins, scale = maxs - mins))
train_ <- scaled[idx == 1, ]
test_ <- scaled[idx == 2, ]
n <- names(train_)
f <- as.formula(paste("y ~", paste(n[!n %in% "y"], collapse = " + ")))
nn <- neuralnet(f, data = train_, hidden = c(5, 3), linear.output = TRUE)
predict_test_nn <- neuralnet::compute(nn, test[, 1:ncol(df) - 1])
predict_test_nn_ <- predict_test_nn$net.result * (max(df$y) - min(df$y)) + min(df$y)
predict_train_nn <- neuralnet::compute(nn, train[, 1:ncol(df) - 1])
predict_train_nn_ <- predict_train_nn$net.result * (max(df$y) - min(df$y)) + min(df$y)

```

```

neuralnet_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict_train_nn)
neuralnet_train_RMSE_mean <- mean(neuralnet_train_RMSE)
neuralnet_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict_test_nn)
neuralnet_test_RMSE_mean <- mean(neuralnet_test_RMSE)

neuralnet_holdout_RMSE[i] <- mean(c(neuralnet_test_RMSE))
neuralnet_holdout_RMSE_mean <- mean(neuralnet_holdout_RMSE)

} # Closing brace for numresamples
  return(neuralnet_holdout_RMSE_mean)

} # closing brace for neuralnet1 function

neuralnet1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamp
#> [1] 4.084896
warnings() # no warnings for individual neuralnet function

```

3.0.18 15. Partial Least Squares

```

library(pls)
#>
#> Attaching package: 'pls'
#> The following objects are masked from 'package:arm':
#>
#>   coefplot, corrplot
#> The following object is masked from 'package:stats':
#>
#>   loadings

# Set initial values to 0
pls_train_RMSE <- 0
pls_test_RMSE <- 0
pls_validation_RMSE <- 0
pls_holdout_RMSE <- 0
pls_test_predict_value <- 0
pls_validation_predict_value <- 0

pls1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamp) {

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y

```

```

y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right

df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
train <- df[idx == 1,]
test <- df[idx == 2,]

pls_train_fit <- pls::plsr(train$y ~ ., data = train)
pls_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = pls_train_fit,
pls_train_RMSE_mean <- mean(pls_train_RMSE)
pls_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = pls_train_fit, ne
pls_test_RMSE_mean <- mean(pls_test_RMSE)

} # Closing brace for numresamples loop
  return( pls_test_RMSE_mean)

} # Closing brace for pls1 function

pls1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 25)
#> [1] 6.013217
warnings() # no warnings for individual pls function

```

3.0.19 16. Principal Components Regression

```

library(pls) # To run pcr models

#Set initial values to 0
pcr_train_RMSE <- 0
pcr_test_RMSE <- 0
pcr_validation_RMSE <- 0
pcr_holdout_RMSE <- 0
pcr_test_predict_value <- 0
pcr_validation_predict_value <- 0

pcr1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

# Set up random resampling
for (i in 1:numresamples) {

```

```

# Changes the name of the target column to y

y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the right

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
train <- df[idx == 1,]
test <- df[idx == 2,]

pcr_train_fit <- pls::pcr(train$y ~ ., data = train)
pcr_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = pcr_train_fit, newdata = train))
pcr_train_RMSE_mean <- mean(pcr_train_RMSE)
pcr_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = pcr_train_fit, newdata = test))
pcr_test_RMSE_mean <- mean(pcr_test_RMSE)

} # Closing brace for numresamples loop
  return(pcr_test_RMSE_mean)

} # Closing brace for PCR function

pcr1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 1000)
#> [1] 6.485505
warnings() # no warnings for individual pls function

```

3.0.20 17. Random Forest

```

library(randomForest)

# Set initial values to 0
rf_train_RMSE <- 0
rf_test_RMSE <- 0
rf_validation_RMSE <- 0
rf_holdout_RMSE <- 0
rf_test_predict_value <- 0
rf_validation_predict_value <- 0

# Set up the function
rf1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples) {
  # Set up random resampling

```

```

for (i in 1:numresamples) {

  #Changes the name of the target column to y
  y <- 0
  colnames(data)[colnum] <- "y"

  # Moves the target column to the last column on the right
  df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column

  # Breaks the data into train, test and validation sets
  idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
  train <- df[idx == 1,]
  test <- df[idx == 2,]

  rf_train_fit <- tune.randomForest(x = train, y = train$y, data = train)
  rf_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = rf_train_fit$bestModel))
  rf_train_RMSE_mean <- mean(rf_train_RMSE)
  rf_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = rf_train_fit$bestModel))
  rf_test_RMSE_mean <- mean(rf_test_RMSE)

} # Closing brace for numresamples loop
return(rf_test_RMSE_mean)

} # Closing brace for rf1 function

rf1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 25)
#> [1] 1.597232
warnings() # no warnings for individual random forest function

```

3.0.21 18. Ridge Regression

```

library(glmnet) # So we can run the ridge model

# Set initial values to 0
ridge_train_RMSE <- 0
ridge_test_RMSE <- 0
ridge_validation_RMSE <- 0
ridge_holdout_RMSE <- 0
ridge_test_predict_value <- 0
ridge_validation_predict_value <- 0
ridge_test_RMSE <- 0
ridge_test_RMSE_df <- data.frame(ridge_test_RMSE)
ridge_validation_RMSE <- 0
ridge_validation_RMSE_df <- data.frame(ridge_validation_RMSE)

```

```

ridge_holdout_RMSE <- 0
ridge_holdout_RMSE_df <- data.frame(ridge_holdout_RMSE)

# Create the function:
ridge1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples) {

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the right
df <- df[sample(nrow(df)), ] # randomizes the rows

# Breaks the data into train, test and validation sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
train <- df[idx == 1, ]
test <- df[idx == 2, ]

# Set up the ridge model
y <- train$y
x <- data.matrix(train %>% dplyr::select(-y))
ridge_model <- glmnet::glmnet(x, y, alpha = 0)
ridge_cv <- cv.glmnet(x, y, alpha = 0)
best_ridge_lambda <- ridge_cv$lambda.min
best_ridge_model <- glmnet::glmnet(x, y, alpha = 0, lambda = best_ridge_lambda)
ridge_test_pred <- predict(best_ridge_model, s = best_ridge_lambda, newx = data.matrix(x))

ridge_test_RMSE <- Metrics::rmse(actual = test$y, predicted = ridge_test_pred)
ridge_test_RMSE_df <- rbind(ridge_test_RMSE_df, ridge_test_RMSE)
ridge_test_RMSE_mean <- mean(ridge_test_RMSE_df$ridge_test_RMSE[2:nrow(ridge_test_RMSE_df)])

ridge_holdout_RMSE <- mean(ridge_test_RMSE_mean)
ridge_holdout_RMSE_df <- rbind(ridge_holdout_RMSE_df, ridge_holdout_RMSE)
ridge_holdout_RMSE_mean <- mean(ridge_holdout_RMSE_df$ridge_holdout_RMSE[2:nrow(ridge_holdout_RMSE_df)])

} # closing brace for numresample
  return(ridge_holdout_RMSE_mean)

} # closing brace for the ridge function

ridge1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 1000)

```

```
#> [1] 5.016598
warnings() # no warnings for individual ridge function
```

3.0.22 19. Robust Regression

```
library(MASS) # To run rlm function for robust regression

# Set initial values to 0
robust_train_RMSE <- 0
robust_test_RMSE <- 0
robust_validation_RMSE <- 0
robust_holdout_RMSE <- 0
robust_test_predict_value <- 0
robust_validation_predict_value <- 0

# Make the function
robust1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right

df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
train <- df[idx == 1,]
test <- df[idx == 2, ]

robust_train_fit <- MASS::rlm(x = train[, 1:ncol(df) - 1], y = train$y)
robust_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = robust_train_fit$fitted.values)
robust_train_RMSE_mean <- mean(robust_train_RMSE)
robust_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = MASS::rlm(y ~ x, data = train), newdata = test))
robust_test_RMSE_mean <- mean(robust_test_RMSE)

} # Closing brace for numresamples loop
return(robust_test_RMSE_mean)

} # Closing brace for robust1 function
```

```
robust1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 1000)
#> [1] 4.986633
warnings() # no warnings for individual robust function
```

3.0.23 20. Rpart

```
library(rpart)

# Set initial values to 0
rpart_train_RMSE <- 0
rpart_test_RMSE <- 0
rpart_validation_RMSE <- 0
rpart_holdout_RMSE <- 0
rpart_test_predict_value <- 0
rpart_validation_predict_value <- 0

# Make the function
rpart1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples) {

  # Set up random resampling
  for (i in 1:numresamples) {

    # Changes the name of the target column to y
    y <- 0
    colnames(data)[colnum] <- "y"

    # Moves the target column to the last column on the right
    df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the right

    # Breaks the data into train and test sets
    idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
    train <- df[idx == 1,]
    test <- df[idx == 2,]

    rpart_train_fit <- rpart::rpart(train$y ~ ., data = train)
    rpart_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = rpart_train_fit))
    rpart_train_RMSE_mean <- mean(rpart_train_RMSE)
    rpart_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = rpart_train_fit, newdata = test))
    rpart_test_RMSE_mean <- mean(rpart_test_RMSE)

  } # Closing loop for numresamples
  return(rpart_test_RMSE_mean)
```



```

} # Closing brace for rpart1 function

rpart1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresamples = 100)
#> [1] 5.014648
warnings() # no warnings for individual rpart function

```

3.0.24 21. Support Vector Machines

```

library(e1071)

# Set initial values to 0
svm_train_RMSE <- 0
svm_test_RMSE <- 0
svm_validation_RMSE <- 0
svm_holdout_RMSE <- 0
svm_test_predict_value <- 0
svm_validation_predict_value <- 0

# Make the function
svm1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresamples){

  # Set up random resampling
  for (i in 1:numresamples) {

    # Changes the name of the target column to y
    y <- 0
    colnames(data)[colnum] <- "y"

    # Moves the target column to the last column on the right
    df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the last column

    # Breaks the data into train and test sets
    idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_amount))
    train <- df[idx == 1,]
    test <- df[idx == 2,]

    svm_train_fit <- e1071::tune.svm(x = train, y = train$y, data = train)
    svm_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = svm_train_fit$bestTune))
    svm_train_RMSE_mean <- mean(svm_train_RMSE)
    svm_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = svm_train_fit$bestTune, newdata = test))
    svm_test_RMSE_mean <- mean(svm_test_RMSE)

  } # Closing brace for numresamples loop
  return(svm_test_RMSE_mean)
}

```

```

} # Closing brace for svm1 function

svm1(data = MASS::Boston, colnum = 14, train_amount = 0.60, test_amount = 0.40, numresam
#> [1] 2.332147
warnings() # no warnings for individual Support Vector Machines function

```

3.0.25 22. Trees

```

library(tree)

# Set initial values to 0

tree_train_RMSE <- 0
tree_test_RMSE <- 0
tree_validation_RMSE <- 0
tree_holdout_RMSE <- 0
tree_test_predict_value <- 0
tree_validation_predict_value <- 0

# Make the function
tree1 <- function(data, colnum, train_amount, test_amount, validation_amount, numresampl

# Set up random resampling
for (i in 1:numresamples) {

# Changes the name of the target column to y
y <- 0
colnames(data)[colnum] <- "y"

# Moves the target column to the last column on the right
df <- data %>% dplyr::relocate(y, .after = last_col()) # Moves the target column to the

# Breaks the data into train and test sets
idx <- sample(seq(1, 2), size = nrow(df), replace = TRUE, prob = c(train_amount, test_
train <- df[idx == 1,]
test <- df[idx == 2, ]

tree_train_fit <- tree::tree(train$y ~ ., data = train)
tree_train_RMSE[i] <- Metrics::rmse(actual = train$y, predicted = predict(object = tree
tree_train_RMSE_mean <- mean(tree_train_RMSE)
tree_test_RMSE[i] <- Metrics::rmse(actual = test$y, predicted = predict(object = tree_t
tree_test_RMSE_mean <- mean(tree_test_RMSE)

```