



Minnesota State University, Mankato  
Cornerstone: A Collection of Scholarly  
and Creative Works for Minnesota  
State University, Mankato

---

All Undergraduate Theses and Capstone  
Projects

Undergraduate Theses and Capstone Projects

---

2020

## Development of Machine Learning Tutorials for R

John Pintar  
*Minnesota State University, Mankato*

Follow this and additional works at: <https://cornerstone.lib.mnsu.edu/undergrad-theses-capstones-all>



Part of the [Software Engineering Commons](#), and the [Theory and Algorithms Commons](#)

---

### Recommended Citation

Pintar, J. (2020). Development of machine learning tutorials for R [Bachelor of Science thesis, Minnesota State University, Mankato]. Cornerstone: A Collection of Scholarly and Creative Works for Minnesota State University, Mankato. <https://cornerstone.lib.mnsu.edu/undergrad-theses-capstones-all/2/>

This Thesis is brought to you for free and open access by the Undergraduate Theses and Capstone Projects at Cornerstone: A Collection of Scholarly and Creative Works for Minnesota State University, Mankato. It has been accepted for inclusion in All Undergraduate Theses and Capstone Projects by an authorized administrator of Cornerstone: A Collection of Scholarly and Creative Works for Minnesota State University, Mankato.

# Development of Machine Learning Tutorials for R

by

John Pintar

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

Bachelor of Science

in

Cognitive Science

with an emphasis in Computer Science

Minnesota State University, Mankato

Mankato, Minnesota

May 7, 2020

## Abstract

Machine learning (ML) techniques developed in computer science have revolutionized nearly every sector of industry. Despite the prevalence and usefulness of ML, students outside of computer science rarely receive training in ML. Students frequently receive training in statistical analysis, often using the software package R, which is free, open source, and has additional downloadable modules. A popular module is the ML package **caret**, which contains 238 different ML algorithms, each with 0-9 hyperparameters. **caret** is powerful, flexible, and provides consistent syntax across algorithms. In the hands of an experienced practitioner, this tunability is welcomed and can increase accuracy. However, when used by a beginning student, the large number of options can become overwhelming and hinder their learning. **babyCaret** is an ML package for R developed in this work to reduce this complexity and support student learning while matching **caret**'s syntax. The goal is to teach users about the application of ML directly inside their familiar R environment. **babyCaret** contains integrated tutorials activated by a function call. These tutorials teach users about the application, interpretation, and technical aspects of four algorithms:  $k$ -nearest neighbors, apriori,  $k$ -prototypes, and decision tree. The  $k$ -nearest neighbors implementation was designed by the author. Decision trees are computed via the **rpart** R package for its visualization capabilities,  $k$ -prototypes uses a modified implementation by Gero Szepannek, and apriori uses the **arules** R package. A limited number of hyperparameters are available for tuning. The rest have either been automated or fixed to their simplest configuration to reduce complexity, which may affect accuracy. **babyCaret** is an open-source teaching tool, a simple and functional beginner ML package, and a stepping-stone to the more complex

**caret**. Evaluation includes runtime comparison between  $k$ -nearest neighbors computed using **caret** and **babyCaret**, and runtime comparison between Szepannek's implementation and our modified version of  $k$ -prototypes. **babyCaret**'s KNN implementation had lower runtime than **caret**'s and the modified version of Szepannek's  $k$ -prototypes implementation had lower runtime than the original. Evaluation of the tutorials involved distributing them to an intelligent systems class as supplemental course content. Tutorials were successfully used in the course setting.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	The R Statistical Package . . . . .	3
2.2	Machine learning . . . . .	4
2.3	<b>caret</b> : an ML module for R . . . . .	10
2.4	Extending R: <b>babyCaret</b> , a new ML module . . . . .	11
<b>3</b>	<b>Tutorial Development</b>	<b>13</b>
3.1	<b>Rcpp</b> . . . . .	13
3.2	Algorithms . . . . .	15
3.2.1	$k$ -nearest Neighbors . . . . .	16
3.2.2	$k$ -prototypes . . . . .	19
3.2.3	Decision Tree . . . . .	21
3.2.4	Apriori . . . . .	26
3.3	Tutorials . . . . .	26
<b>4</b>	<b>Results</b>	<b>33</b>

4.1	Tutorial Distribution . . . . .	33
4.2	$k$ -nearest Neighbors Runtime . . . . .	34
4.3	$k$ -prototypes Runtime . . . . .	35
<b>5</b>	<b>Conclusions &amp; Future Work</b>	<b>36</b>
5.1	Assessing Implementation Performance . . . . .	36
5.2	Future work . . . . .	37
5.2.1	User Evaluation . . . . .	37
5.2.2	RStudio 1.3 . . . . .	38
5.2.3	Cross Validation . . . . .	38
5.2.4	KNN . . . . .	39
5.2.5	$k$ -prototypes . . . . .	39
5.2.6	Apriori . . . . .	39
<b>A</b>	<b>babyCaret's tutorials</b>	<b>45</b>
A.1	What is <b>babyCaret</b> ? . . . . .	45
A.2	How to Operate <b>babyCaret</b> 's Tutorials . . . . .	45
A.3	Prerequisite R Programming . . . . .	46
A.4	Creating Training and Testing Sets . . . . .	51
A.5	Using the Algorithms (in general) . . . . .	54
A.6	$k$ -nearest Neighbors algorithm . . . . .	58
A.7	$k$ -prototypes algorithm . . . . .	64
A.8	Apriori algorithm . . . . .	69
A.9	Decision tree algorithm . . . . .	72

<b>B Materials for Future Formal User Feedback</b>	<b>78</b>
B.1 Help Sheet . . . . .	78
B.2 User Survey . . . . .	80
B.3 Researcher Instructions . . . . .	87
<b>C Public Functions</b>	<b>89</b>

# Table of Figures

2.1	Base R Command Line Interface . . . . .	4
2.2	RStudio Integrated Development Environment . . . . .	5
3.1	RStudio Console Window showing a Tutorial Section Containing Information on Manhattan Distance . . . . .	18
3.2	A Decision Tree using <i>Species</i> as the target attribute on Fischer’s Iris Dataset [8] . . . . .	22
3.3	Regression Tree Showing Interior Node Values [8] . . . . .	24
3.4	Classification Tree on Fischer’s Iris <i>not</i> Showing Interior Node Values [8] . .	25
3.5	<b>babyCaret’s</b> Tutorial Menu . . . . .	27
3.6	Multiple Choice Question about the Apriori Algorithm . . . . .	29
3.7	True/False Question about the Apriori Algorithm . . . . .	29
3.8	Programming Question Related to the <i>k</i> -prototypes Algorithm . . . . .	30



# List of Tables

2.1	Categories of ML Algorithms included in <b>babyCaret</b> . . . . .	7
4.1	Mean KNN runtime in milliseconds on Fisher's Iris dataset with $k = 5$ . . . .	34
4.2	Mean $k$ -prototypes runtime in milliseconds on Fisher's Iris dataset with $k = 5$ , <code>nstart = 1, iter.max = 100</code> . . . . .	35

# Chapter 1

## Introduction

Machine learning (ML) techniques developed in computer science have revolutionized nearly every sector of academia and industry. Despite the prevalence and usefulness of ML, students outside of computer science rarely receive training in ML.

The goal of this thesis is to develop a tutorial module that allows novice users to both learn about and implement machine learning inside of a commonly used statistical software package. Students frequently receive training in statistical analysis, often using the software package R [21], which is free, open source, and allows for open development of add-on modules. R currently has ML capabilities through a module, **caret** [15].

Learning Challenges:

1. When learning a new analysis software package, it is easy to only learn the bare minimum required to complete an analysis.
2. Analysis packages are often complex.
3. Important details of an analysis technique need to be understood.

This work's contribution is a freely available add-on ML module developed to extend R. This module is named **babyCaret**. **babyCaret** is a simplified ML module with syntax inspired by the existing module **caret** that allows users to implement four machine learning

algorithms. Unlike other modules, **babyCaret** contains console-based tutorials to teach users about these algorithms.

In chapter 2, this thesis presents background information on machine learning and R, the software package our module has been developed for. In chapter 3, the development of the tutorial and ML module **babyCaret** is discussed. Chapter 4 includes the results of distributing this module to a class of intelligent systems students as supplemental course material as well as comparative runtime analysis.

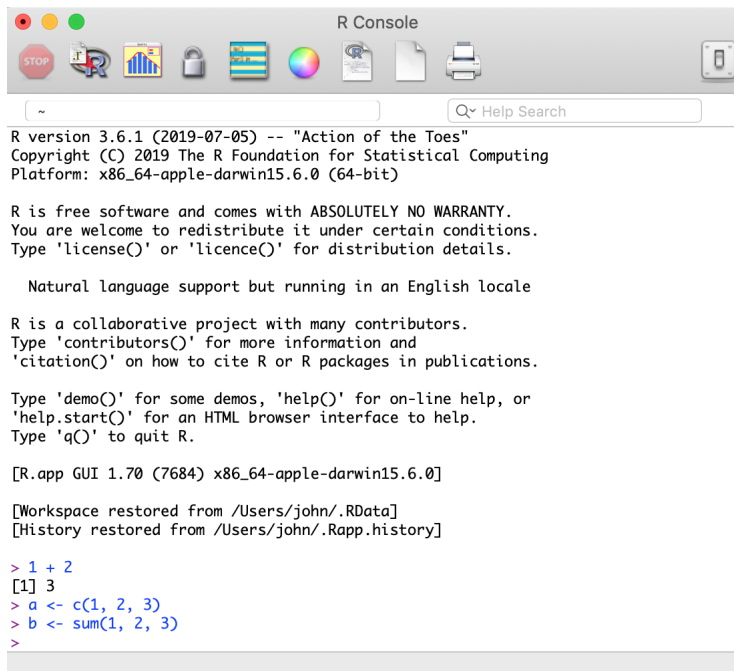
## Chapter 2

### Background

In this chapter, the necessary background for this work is discussed. This covers the statistical software package, R, a brief overview of machine learning including the four algorithms used by **babyCaret**, and lastly, discussion of **babyCaret** and its tutorials.

#### 2.1 The R Statistical Package

R is a statistical software package and programming language. It is used extensively throughout both industry and academia and is one of the leading analysis packages for science in general. R was chosen as the context for this work for multiple reasons. The first reason is that R is commonly used for ML. The second reason is its diverse user base; R is not a language which caters specifically to computer scientists. Its user base includes statisticians, financial analysts, and scientists of various disciplines. Second is its long history. R has been available for over 20 years [13] and is an offshoot of an even older language, S. The final reason is that it is free and open source. Users have access to the code used to create R and are able to modify that code. This allows for development of modules that can be easily and freely distributed—including the source code used to create it. Much of R’s functionality comes from open source development of add-on modules. These modules are frequently created by developers outside of the R Core Team and can be hosted on multiple repositories



```

R Console
~
Q Help Search
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.70 (7684) x86_64-apple-darwin15.6.0]

[Workspace restored from /Users/john/.RData]
[History restored from /Users/john/.Rapp.history]

> 1 + 2
[1] 3
> a <- c(1, 2, 3)
> b <- sum(1, 2, 3)
>

```

Figure 2.1: Base R Command Line Interface

for easy installation by users. **babyCaret** is one such module. Beyond modules, there are open source development environments. RStudio [25] is a popular development environment for R which adds graphical user interface (GUI) functionality beyond the standard command line interface. RStudio can be used as desktop or cloud software as RStudio Desktop and RStudio Cloud [23], respectively. Figure 2.1 shows the standard R command line interface, while Figure 2.2 shows RStudio and its GUI.

## 2.2 Machine learning

Machine learning algorithms are algorithms whose results improve as they process data. This happens because of a training process where the algorithm is presented with data. Usually, this learning involves identifying frequent or important patterns. During training,

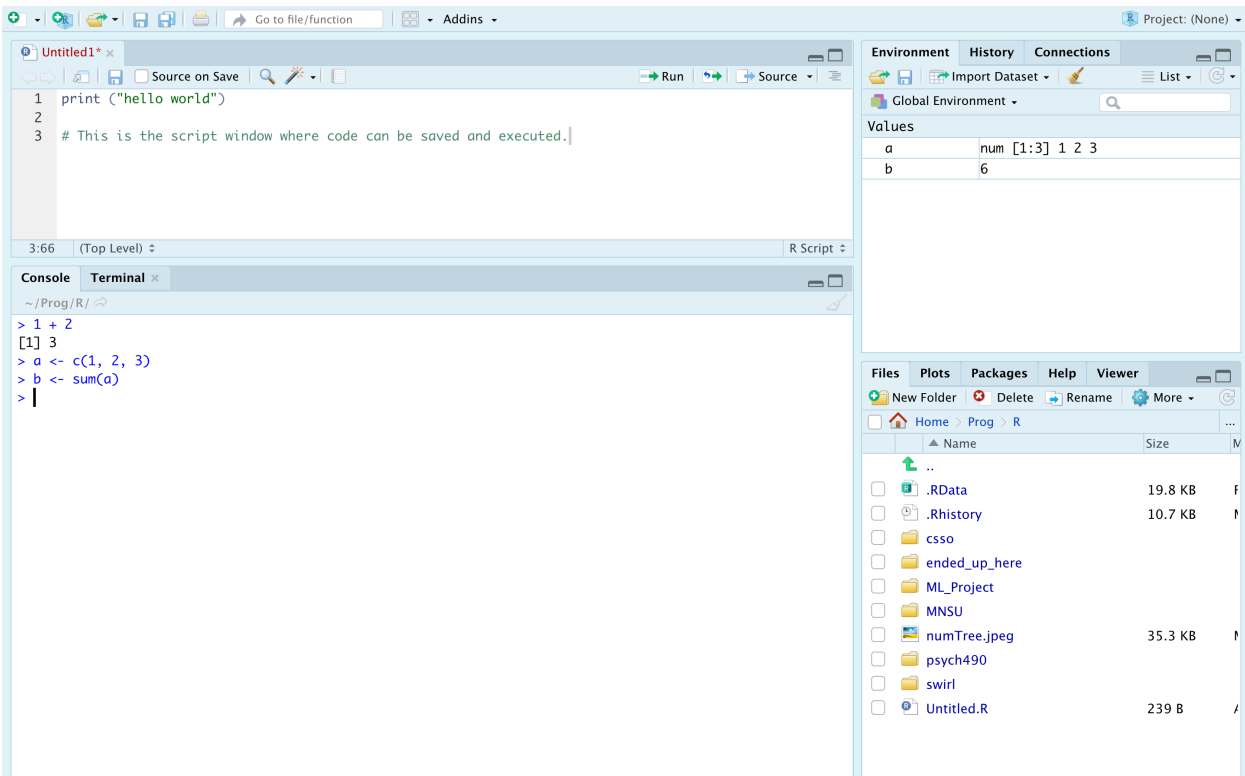


Figure 2.2: RStudio Integrated Development Environment

the algorithm creates a model. The model is then used for classification or regression (two types of prediction), or informing the user of the patterns found in the data through clustering or rule learning. Throughout this paper we will often refer to this broad class of processing—which occurs after model training—as a testing phase or testing process. When the testing phase is prediction, a target variable or attribute must be specified. This target variable specifies what is being predicted.

In the context of this thesis, the data being processed is assumed to be in the form of a data table, or a “data frame” in the syntax of R. Each row in the table is an instance, or a single observation. Each column is an attribute. An instance will have a numeric or categorical value for each attribute. A single attribute may contain either numeric values or categorical values, not both.

When training a model, the user provides the model with hyperparameters, parameters set by the user and used to shape the model. This is in contrast to model parameters learned by the algorithm. In this work, *parameter* is synonymous with hyperparameter, which is the term used in the R module **caret**. The basic level of user ML workflow considered here involves selecting an appropriate algorithm, optimizing the model by tuning parameters, and finally, using the model for prediction and/or data analysis.

The four algorithms implemented in **babyCaret** were chosen due to their ease of interpretation, relevance, flexibility, and capabilities on mixed datasets (i.e., with both numeric and categorical attributes). These are  $k$ -nearest neighbors (KNN),  $k$ -prototypes [12], decision tree [4], and apriori [3] and are described in Table 2.1. Together, these four algorithms enable **babyCaret** to be capable of predicting numeric values (regression), predicting categorical values (classification), identifying groups of similar datapoints (clustering), and finding statistical if/then rules which explain what set of values we can expect to find if we have already

Table 2.1: Categories of ML Algorithms included in **babyCaret**

Category	Algorithm	Testing process
Classification	$k$ -nearest Neighbors, Decision Tree	Predicts missing categorical values
Regression	$k$ -nearest Neighbors, Decision Tree	Predicts missing numeric values
Clustering	$k$ -prototypes	Provides data points with a label based on their similarity to other data points
Association Rule Learning	Apriori	Identifies statistical if-then rules describing co-occurrence of values in the dataset

found a given set of values (association rule learning). Being capable of these four processes allows **babyCaret's** tutorials to cover a wide range of common ML tasks.

$k$ -nearest neighbors is a simple classification and regression algorithm and has been used to identify job satisfaction and organizational commitment as predictors of action identification level [33].  $k$  is the one hyperparameter set by the user. During the training phase, KNN simply stores the entire training set in memory. During the testing phase, KNN calculates a distance or similarity measure between each instance found in the training set and all instances which are having values predicted. That information is used to make predictions. When KNN makes a prediction, the mean (numeric data) or mode (categorical data) of the  $k$  training instances closest or most similar to the testing instance is used as the predicted value. KNN is capable of handling numeric and categorical data.

Decision trees are used for classification and regression in a wide variety of application areas including speech synthesis [5] and recognition [2]. An ensemble of decision trees, or



“random forest,” has been found to be an effective method for imputing missing values in psychological datasets [9]. Decision trees operate by recursively splitting data into maximally homogeneous subsets as measured by homogeneity of the target value where target values are those belonging to the attribute being predicted. The rules used to create these subsets along with their associated target variable means or modes are used when making a prediction. The parameters available for shaping the model are `minSplit`, which sets the minimum number of instances that can exist in a subset for a split to be attempted and `maxDepth`, which sets the maximum number of splits a subset can be away from the original complete dataset. These two parameters are used to reduce the number of rules and subsets created by the model, a process known as pruning. In the context of the decision tree algorithm, this can avoid overfitting the model to the training data. Overfitting results in a model that performs well on the training set, but poorly on others. The decision tree algorithm is capable of handling numeric and categorical data.

The  $k$ -prototypes algorithm is used for clustering data into groups of similar data and has been used to analyze user patterns from the video game platform, STEAM [22]. Clustering in general is used for exploratory data analysis and for assigning categorical values to a new attribute corresponding to which cluster the datapoint belongs to. During training,  $k$ -prototypes first randomly chooses four existing datapoints to be used as prototypes. A prototype is the most representative datapoint (existing in the dataset or not) of a cluster. Each datapoint has its similarity or distance to each cluster calculated. Each datapoint is then assigned to its nearest prototype. The prototypes are then recalculated for each cluster. This process stops when the prototypes converge. When using the model to label data, the label corresponding to the instance’s closest prototype is applied.  $k$ -prototypes is an amalgamation of two other algorithms,  $k$ -means and  $k$ -modes.  $k$ -means is only capable of

handling numeric data and  $k$ -modes can only handle categorical data. As the combination of these two,  $k$ -prototypes is capable of handling numeric and categorical data, because  $k$ -prototypes uses the distance function from  $k$ -means on numeric data and the distance function from  $k$ -modes on categorical data.

Apriori is used for association rule learning and frequent itemset mining. An example of a frequent itemset is, “genes A, B, C, and D are frequently found together.” A frequent itemset can then be further processed into association rules such as, “gene A and B strongly suggest the presence of gene D.” Apriori has been used to find associations between mutated cancer genes. The algorithm allowed researchers to identify associations between up to four genes, when the previous maximum was two [11]. Apriori uses an iterative process which generates candidate frequent itemsets of size  $N$ . Apriori uses two stages of pruning to identify *infrequent* itemsets. The first stage of pruning uses the downward closure property, commonly referred to as the apriori principle. The downward closure property states that all size  $N - 1$  subsets of a frequent itemset must also be frequent. All candidate itemsets satisfying the downward closure property then have their frequency directly calculated from the dataset previous to the next iteration. Those that do not meet minimum frequency are pruned. Once frequent itemsets have been found, they can be used to find unidirectional if-then relationships between itemsets. Three parameters control apriori model creation: `minSup` defines the rate of occurrence an itemset must have, or “support,” to be considered frequent, `minConf` defines the minimum relation strength a rule must have, or “confidence” to be returned to the user, and `maxLen` sets the maximum number of items that may be contained in a frequent itemset. Unlike the previous three algorithms, apriori is not able to be used directly on mixed data without discretizing numeric values or reassigning numeric values to categorical values in a way that reduces their meaning. Apriori must discretize

numeric values. Association rule learning algorithms fundamentally operate on discrete values so the ability to operate on mixed datasets was not expected out of this category of algorithm.

Implementation details for all four algorithms are discussed in chapter 3 and further details about how the algorithms work, as presented in the tutorials, are in Appendix A.

## 2.3 **caret: an ML module for R**

**caret** is a popular machine learning module for R that extends the base functionality of R to allow a user to implement 238 ML algorithms. **caret** solves the problem of integrating many disjoint implementations, as many implementations used by **caret** are available to R users in modules separate from **caret**. Some of these modules include a single implementation, while others include multiple implementations. It can be difficult for users to keep track of differing workflows between these disjoint modules. **caret** interfaces with these modules and imposes a single workflow on all of them.

**caret** has the ability to automatically tune . This does not require the user to know what those parameters do in order to improve the training of a model. A user does not have to know what any parameter does in order for them to improve a model. **caret** can select the optimal value on its own through cross validation. This is done by repeatedly training and testing models with different parameter on resampled training data. Each set of parameters receives an accuracy score. The set that results in highest accuracy is used as the final set of parameters for the algorithm.

**caret**'s only unique implementation is its KNN implementation [16]. The rest come from other modules. **babyCaret**'s KNN implementation is also unique to **babyCaret**. **caret**

uses the **rpart** R package [28] to train decision trees, as does **babyCaret**. **caret** does not have clustering or association rule learning capabilities, so it does not contain  $k$ -prototypes or apriori.

**caret**'s workflow for the ML training and testing process could be simplified for users new to ML. **caret** allows up to 9 parameters to be tuned for each of its 238 algorithms, and these can be tuned automatically, hiding the functionality from new users. **caret** can be used in RStudio cloud, RStudio desktop, and R's command line interface.

## 2.4 Extending R: **babyCaret**, a new ML module

Extending R to better support new users can be done by maintaining syntax and function names, by reducing parameters available for tuning, and requiring manual tuning of these parameters. **babyCaret**, the module developed here, uses 1-3 parameters instead of 0-9. Requiring manual tuning requires the user to understand how the parameters affect the model as well as encourages experimentation. One goal in developing **babyCaret**'s machine learning capabilities was to be as fast and simple as possible. Speed ensures that the interactive tutorials do not require the user to wait for processing. Simplified algorithm implementations allow for their true functioning to be more easily explained during tutorials.

Along with ML capabilities, **babyCaret** provides integrated ML tutorials. Two existing tutorial modules are **learnr** [26] and **swirl** [14]. **learnr** allows for the development of feature rich tutorials containing elements such as interactive graphs, videos, and programming exercises. These tutorials are then hosted online with the main repository being RStudio's [shinyapps.io](https://shinyapps.io) [24]. **learnr** was not used in this work because it does not currently run inside the user's R environment. **swirl** is a tutorial package that runs in the user's R console. It

was one of the main sources of inspiration for **babyCaret**. **swirl** has one tutorial section which covers clustering, but no other tutorials on ML algorithms. Developers are able to write tutorials for the **swirl** platform, but those tutorials are then run through **swirl**. **swirl** does not allow for tutorials to be embedded into an ML module, and externally developed tutorials must be searched for and installed by the user. Also, unlike **babyCaret**, **swirl** has no direct ML capabilities. Integrating ML tools and ML tutorials into one module allows for the tool aspect to be tailored to the tutorial aspect. The tool aspect can be simplified to support tutorials targeting new users rather than creating tutorials which are forced to deal with the complexity of existing tools.

Beyond learning about ML, **babyCaret** can support learning of the R environment. **babyCaret**'s integrated tutorials run directly inside the user's R environment. To support user learning, key aspects of the tutorials are the minimum R programming required to use **babyCaret**, how to split a single dataset into training and testing sets, how to implement the four included algorithms, and how the four algorithms operate. Student learning is fostered through processes traditionally used in teaching and reinforcement of human learning, such as descriptive text, multiple choice questions, true/false questions, and programming questions to help the users view key content and retain it for later use. Details on development are discussed in the next chapter.

**babyCaret**'s tutorials are intended to be used with the RStudio environment. The tutorials developed here assume that the user is working inside of RStudio and inform the user about features specific to RStudio.

## Chapter 3

### Tutorial Development

This chapter discusses the development of **babyCaret**. **babyCaret** has been programmed in both the R and C++ programming languages. The interaction between languages is mediated by R’s **Rcpp** package [7], which is described below. Then, the implementations for the four ML algorithms used by **babyCaret** are reviewed. Finally, the development of **babyCaret**’s interactive tutorials is discussed. The content of **babyCaret**’s tutorials can be found in Appendix A.

#### 3.1 Rcpp

The choice to extend **babyCaret** with a language other than R allows more options for development as well as the use of code which steps outside of the typical R paradigm. Combining R and C++ specifically was done for two reasons. The first is because C++ is the most common language used to extend R. There are significant resources and support available for doing so; Dirk Eddelbuettel’s R package, **Rcpp** [7] is a stable and established R package that **babyCaret** uses when integrating R and C++. The second reason is that C++ excels in computational performance where R does not.

In professional ML software, runtime is a consideration. When training complex models on large datasets, the runtime can extend beyond time available for user learning and

tutorials. **babyCaret** was developed for use by novices, so large scale datasets are not expected. However, runtime is a priority for a separate reason. **babyCaret's** integrated tutorials should feel snappy and responsive to the user. Short runtime is important to create a satisfying user experience during the tutorials. Also, shorter runtime gives the program headroom so that additional features can be added without a noticeable increase in runtime.

Using **Rcpp**, a C++ function can be written directly inside of an R script using `cppFunction()`. “Upon calling the `cppFunction()` ...the C++ code is both compiled and linked, and then imported into R under the name of the function supplied” [6]. **Rcpp** contains a set of C++ classes which are analogous to common R classes. For example, a character matrix in R has an **Rcpp** counterpart. **Rcpp** handles conversion between C++ objects and R objects. Some native C++ classes are supported. For example, an R one-element numeric vector can be passed to a C++ function and will be converted by **Rcpp** into a C++ double precision float. The conversion works bidirectionally. If that object is then returned by the C++ function, it will be converted into an R one-element numeric vector. While this is useful for small components of code, it also supports use of larger or complete segments of code.

Iteration is a key structure where R performance suffers. In many cases, R must create an entirely new object for each iteration of a loop. For example, if a vector is being appended 100 times though the use of a loop, R will initialize 100 vectors. In some cases, this be addressed by treating R more like a lower level programming language and initializing an object to its desired size prior to looping [18]. Additionally, looping can be slow in R due to being an interpreted language. During each iteration of the loop, R has to determine what class it is dealing with and then look up the methods for that class [30]. Neither looping nor dealing with classes decrease C++'s speed. C++ can be used as a high performance replacement for an entire R script or can be used to replace slow sections of an R script.

When iteratively computing a sum, doing so with a C++ function imported into R by **Rcpp** has been shown to be approximately nine times faster than the equivalent R function at  $4.04\mu s$  vs.  $36.71\mu s$  [31]. The same test also showed memory allocation to be approximately 79 times higher when using the R function at 2.49KB vs. 187.5KB [31]. When ultra fast runtimes and low memory usage are required, which makes complex ML algorithms usable, C++ is a better choice than R. Using **Rcpp** allows **babyCaret** to take advantage of C++'s high performance while keeping the user-accessible functions entirely in R's interface.

## 3.2 Algorithms

**babyCaret** uses four ML algorithms:  $k$ -nearest neighbors,  $k$ -prototypes, decision tree, and apriori. These algorithms were chosen with simplicity and flexibility in mind. The main criteria for flexibility was being able to work on mixed data i.e., with both numeric and categorical values. We did not want the number of algorithms to grow simply because more were required to make **babyCaret** fully compatible with mixed data. All algorithms are natively compatible with mixed data except apriori, which discretizes numeric values to simulate categorical values.

Algorithms were made appropriate for **babyCaret**'s tutorials through different approaches for each algorithm. **babyCaret**'s KNN implementation is unique to **babyCaret** and uses a combination of R and C++. A unique implementation ensures that only the necessary overhead required to integrate with the tutorials would be present in the implementation. Using C++ for computationally intensive operations such as distance calculation was done in an attempt to reduce runtime.

$k$ -prototypes uses a modified version of an existing implementation by Gero Szepannek



[27]. C++ was used for distance calculations instead of the original R code, and minor changes were made to increase flexibility, for example, enabling use on homogeneous datasets containing only numeric or only categorical attributes

The decision tree algorithm uses an existing R ML module, **rpart** [28]. Although **rpart** is a module with many options, **babyCaret** disables some of **rpart**'s advanced functionality such as automated model optimization. The philosophical approach imposed by **babyCaret** is that **babyCaret**'s default decision tree model is overly complex and overfits to the training data leading to poor predictions. This then forces the user to understand and use the two pruning parameters left available in order to reduce complexity while increasing predictive accuracy.

Apriori uses the existing R module **arules** [10]. **arules**' apriori implementation does not expose the user to extreme complexity or contain advanced functionality which reduces the value of experimentation for optimization. The output has been forced into a form useful for the tutorials, i.e., a data table sorted so that high-value rules derived from frequent itemsets are in decreasing order.

The design decisions and implementation for these four algorithms are described in more detail below.

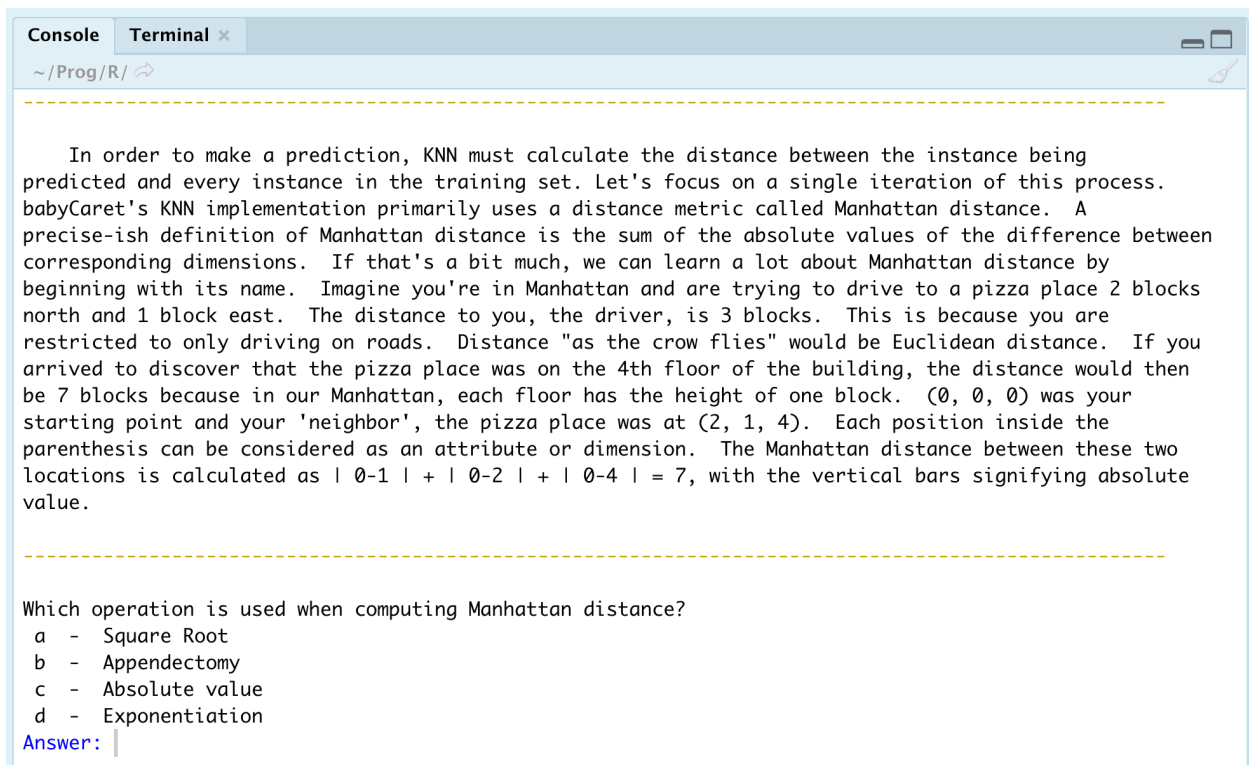
### 3.2.1 *k*-nearest Neighbors

**babyCaret**'s *k*-nearest neighbors (KNN) implementation uses a mix of C++ and R. Training is done entirely in R. An object representing the model is returned containing the training set as an R data frame, the target variable as a vector, the class of the target variable, and the user's selection of *k*, which is this implementation's only parameter.

C++ is used to scale numeric values between zero and one using max/min scaling on both the training set and the testing set during the prediction phase before distances are calculated. Scaling is done during the prediction phase to allow new data to be scaled according to the same maximum and minimum values in the training set. To improve this in future versions, the training data could be scaled during the training phase, with the maximum and minimum values of each attribute recorded so that the new data could be scaled between these values during the prediction process. This would save a small amount of time in the prediction process. Although the runtime for one train-predict cycle would be the same, reducing prediction time is of higher value because the same trained model can be used to make predictions multiple times whereas training is a one time cost.

For the prediction phase, a distance matrix is calculated in C++. The distance matrix contains the distances between every training set instance and every testing set instance. For numeric values, Manhattan distance is used. The Manhattan distance between two points is the distance required to move between the points when traversal must be parallel to an axis. Figure 3.1 shows the description of Manhattan distance as given in the tutorials. Manhattan distance was originally chosen over Euclidean distance because it was believed to be easier to meaningfully explain to the user than Euclidean distance, particularly when under consideration in high dimensional spaces. Its standing as KNN's distance function is further supported by research showing it often outperforms Euclidean distance when used in KNN classification [20]. Additionally, Euclidean distance is used in the  $k$ -prototypes tutorial implementation which provides tutorial information on two distance functions.

When dealing with mixed datasets, ordinal data is converted to an integer before scaling, scaled, then treated as numeric data. Nominal data uses simple matching distance. If there is a match between nominal attributes, the distance contributed is zero. If there is not a match,



The image shows a screenshot of an RStudio Console window. The window title is "Console Terminal x" and the current directory is "~/Prog/R/". The text in the console is as follows:

-----

In order to make a prediction, KNN must calculate the distance between the instance being predicted and every instance in the training set. Let's focus on a single iteration of this process. babyCaret's KNN implementation primarily uses a distance metric called Manhattan distance. A precise-ish definition of Manhattan distance is the sum of the absolute values of the difference between corresponding dimensions. If that's a bit much, we can learn a lot about Manhattan distance by beginning with its name. Imagine you're in Manhattan and are trying to drive to a pizza place 2 blocks north and 1 block east. The distance to you, the driver, is 3 blocks. This is because you are restricted to only driving on roads. Distance "as the crow flies" would be Euclidean distance. If you arrived to discover that the pizza place was on the 4th floor of the building, the distance would then be 7 blocks because in our Manhattan, each floor has the height of one block.  $(0, 0, 0)$  was your starting point and your 'neighbor', the pizza place was at  $(2, 1, 4)$ . Each position inside the parenthesis can be considered as an attribute or dimension. The Manhattan distance between these two locations is calculated as  $|0-1| + |0-2| + |0-4| = 7$ , with the vertical bars signifying absolute value.

-----

Which operation is used when computing Manhattan distance?

- a - Square Root
- b - Appendectomy
- c - Absolute value
- d - Exponentiation

Answer: |

Figure 3.1: RStudio Console Window showing a Tutorial Section Containing Information on Manhattan Distance

the distance contributed is one. One problem with this approach is that it is inflexible. On some datasets, it is possible that two nominal values being mismatched between instances does not contribute as much dissimilarity as one of their nominal attribute's values being very distant. In the current system, they are considered equally dissimilar. One way to introduce some flexibility would be to multiply all matching distances by a user selected parameter. This would allow them to increase or decrease the dissimilarity contribution of mismatched nominal attribute values. This will be considered for future work.

The final stages of the prediction process are handled by R. The distance matrix is recreated to keep only the  $k$  nearest or most similar instances for each testing set instance. If the target variable is numeric, the mean of the  $k$  nearest training instances for each testing instance is assigned as the testing instance's value. If the target variable is categorical, the mode is used. `babyPredict()`, which is outlined in Appendix C, then returns an R data frame object complete with predicted values. The description of KNN given in **babyCaret's** tutorials can be found in Appendix A.6.

### 3.2.2 $k$ -prototypes

Szepannek's  $k$ -prototypes implementation follows the procedure described in Section 2.2. Distance is calculated using Euclidean distance for numeric attributes and the same simple matching distance used by **babyCaret's** KNN implementation for categorical attributes. Unique to this specific implementation is a `lambda` parameter that multiplies the matching distance used for categorical attributes. This allows the a user to set the amount of distance contributed by a mismatch. The distance contributed is equal to `lambda` which has a default value of 1. The other parameters are `iter.max` which ends processing if convergence has not

occurred within its value of iterations and `nstart` which dictates the number of times the algorithm is rerun with new initial random prototypes. Both Szepannek’s implementation and **babyCaret**’s modified version allow for tuning of the same parameters.

The first modification made for **babyCaret** was rewriting distance calculations from R into C++ as using **Rcpp** to reduce runtime was suggested by Szepannek [27]. The approach taken was to use one matrix for numeric variables and another for categorical variables, calculate their respective distances and then merge the two into a final distance matrix.

Szepannek’s implementation includes three stopping conditions which reduce its flexibility. These have been removed. One of these conditions exists to ensure `lambda` is greater than or equal to zero for at least one variable. This check has been removed because setting `lambda` equal to zero is an easy way for a user to completely ignore categorical variables without having to remove them from the dataset. It is assumed the error check exists to encourage the user to use *k*-means if they are not going to consider categorical attributes. This would reduce redundant computation, but **babyCaret** values the reduction of user-experienced complexity over the reduction of redundant computation. Also, being able to reduce `lambda` to 0 allows the user to better understand `lambda`’s role in the algorithm, meeting a learning objective for the tutorial.

The other two removed error checks are mirror images of each other. One stops the training process if there are no numeric variables because that suggests the use of use *k*-modes. This suggestion is printed by Szepannek’s implementation. The other does the same for categorical variables and *k*-means. When there are no numeric variables, *k*-prototypes essentially operates as *k*-modes. When there are no categorical variables, it is essentially *k*-means. Removing these error checks change Szepannek’s implementation into a flexible and novice-friendly implementation capable of clustering many datasets. The description of

$k$ -prototypes given in **babyCaret**'s tutorials can be found in Appendix A.7.

### 3.2.3 Decision Tree

**rpart** is the standard R package for calculating decision trees. Both **caret** and **babyCaret** use **rpart** and contribute a standardized interface and workflow through which the user interacts with this existing established ML package. **rpart** allows significant control over decision trees, and **babyCaret** exerts control over **rpart** in a way to encourage user experimentation and manual optimization to support user learning of the algorithm.

**rpart**'s decision tree implementation recursively splits a dataset by searching for a yes/no rule involving a non-target attribute which will partition the target values into maximally homogeneous subsets. If a regression tree is being built (numeric target attribute), an analysis of variance (ANOVA) technique is used to measure homogeneity. If a classification tree is being built (categorical target attribute), the Gini index is used as the measurement of homogeneity. The Gini index attempts to quantify the distribution of resources, in this case, target values. A Gini index of 0 indicates equal distribution of resources while a Gini index of 1 indicates maximally unequal distribution of resources. When used as the splitting metric for a decision tree, the split leading to the highest Gini index is chosen. The Gini index is equal to 1 minus the sum of squared target class probabilities and is commonly used in economics to quantify income inequality [17]. In the Gini index formula below,  $n_c$  is the number of unique target values or classes in the set, and  $P(C_i)$  is the probability of a randomly chosen value being a member of the  $i$ th unique class.

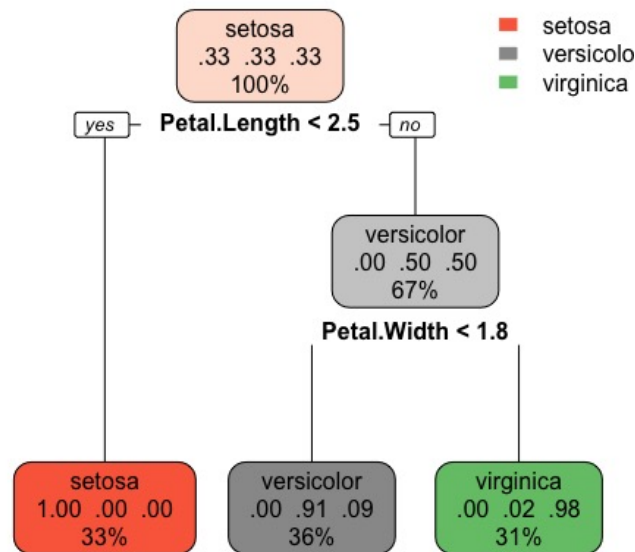


Figure 3.2: A Decision Tree using *Species* as the target attribute on Fischer’s Iris Dataset [8]

$$Gini = 1 - \sum_{i=1}^{n_c} P(C_i)^2.$$

In Figure 3.2, a decision tree was trained on Fischer’s Iris dataset [8]. Fisher’s Iris is a dataset of iris flowers and contains four numeric attributes: petal length, petal width, sepal length, and sepal width. The lone categorical attribute is species, of which there are three possible values: setosa, versicolor, and virginica. Each node in Figure 3.2 represents a subset and shows (from top to bottom) the most common species in the subset, the ratio of setosa, versicolor, and virginica flowers in left to right order, and at the bottom, the percentage of instances from the original dataset contained in the subset. The chosen rule for a split is shown below the node. Observing the ratio of species contained in each node, it can be

seen that the tree is attempting to create maximally homogeneous subsets. Since this is a classification tree, the Gini index is its metric for homogeneity, which is not shown, but can be inferred from the distribution of species at each node.

By default, **babyCaret** will train a full tree. A full tree is a maximally complex (as allowed by the algorithm) tree and runs the risk of overfitting to the training data. This growth is allowed by blocking **rpart**'s cross validation and therefore the pruning that occurs in that stage. Secondly, **rpart** has a parameter called `cp`, or complexity parameter. Any split that does not improve the fit by `cp` is not attempted [28]. **babyCaret** fixes `cp` at -1 to allow for training of a complete tree. Tutorial users are encouraged to explore tree growth by changing other parameters described below.

Surrogate splits used for prediction in the presence of missing data are not addressed since **babyCaret**'s tutorials do not deal with missing attribute values. Not searching for surrogates reduces runtime by 50% [28]. Competitor splits are not found or displayed in order to keep excess information away from the user (again, to focus user learning on the process of decision tree building). This decision may be reevaluated in the future. Although **rpart** is capable of identifying the splitting method, **babyCaret** provides this explicitly. ANOVA splitting is used when the target variable is numeric. Class splitting is used when the target variable is categorical. Lastly, if class splitting is being used, **babyCaret** sets the the measurement of homogeneity to be the Gini index. The description of the decision tree algorithm given in **babyCaret**'s tutorials can be found in Appendix A.9.

The parameters left available for tuning in the tutorial are maximum tree depth, `maxDepth`, and minimum node size a split will be attempted on, `minSplit`. These allow the user to reduce the complexity of the tree and improve the model in a way that is easy to understand and visual in nature. To display a decision tree plot, **babyCaret** has a function called



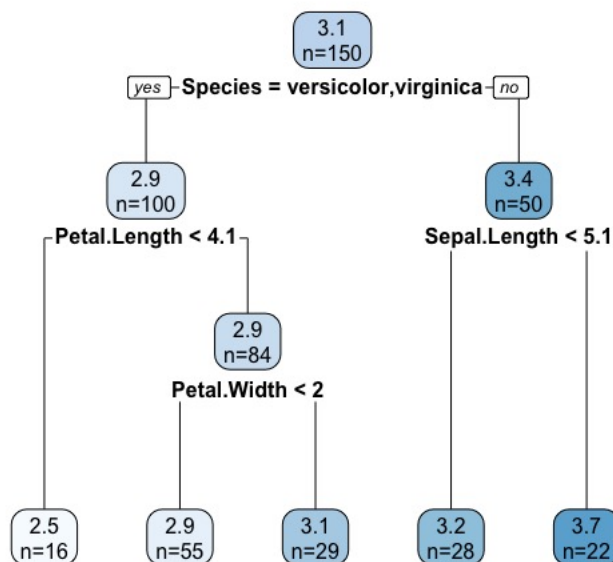


Figure 3.3: Regression Tree Showing Interior Node Values [8]

`plotTree()`. This function calls `rpart.plot()` from an R module of the same name and sets its visualization parameters based on the type of tree model given as an argument, either classification or regression [19]. Additionally, the user can specify whether or not they would like to see interior node values through the `showInterior` parameter. Output from `plotTree()` is shown in Figures 3.2, 3.3, and 3.4. An outline of `plotTree()` can be found in Appendix C.

Figure 3.3 shows a regression tree where the numeric target value is sepal width. Each node shows the mean sepal width in its subset followed by the number of instances in that subset. This is what would be shown if a user set `plotTree()`'s `showInterior` parameter to `TRUE` while plotting a regression tree.

Figure 3.4 shows a classification tree where the categorical target value is species. The nodes show the most numerous target value contained in the subset followed by how many

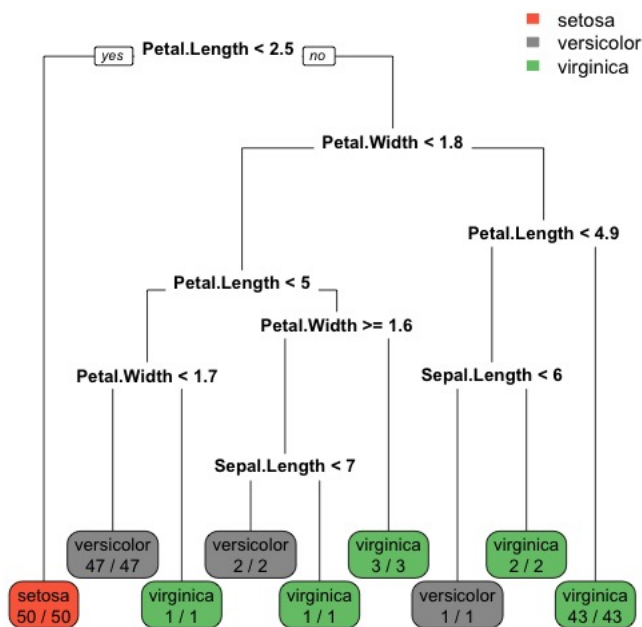


Figure 3.4: Classification Tree on Fischer's Iris *not* Showing Interior Node Values [8]

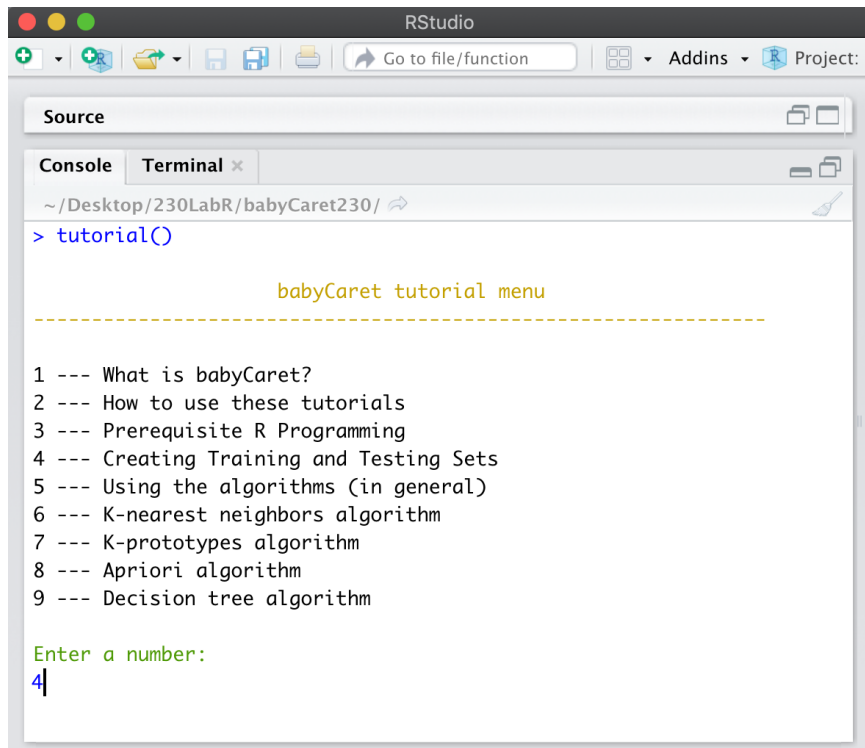
instances have that target value over the number of instances in the subset. The only nodes shown are the final subsets which are not split. These are the subsets which would be used for prediction. Figure 3.4 is an example of what would be shown if a user set `plotTree()`'s `showInterior` parameter to `FALSE` while plotting a classification tree. This is a fully grown tree and shows that when the decision tree algorithm is unrestricted, homogeneity can come at the price of increased model complexity. Six out of the nine subsets contain three or fewer instances. This is an example of an overfitted decision tree model; it would likely make less accurate predictions than a more generalized model. One of the tasks in **babyCaret**'s tutorials is to start with a classification tree resembling Figure 3.4 and use pruning techniques to make the model appear more like Figure 3.2

### 3.2.4 Apriori

**babyCaret** runs the apriori algorithm using R's **arules** package [10]. **arules** did not require the complexity reduction that **rpart** did. Users can modify all control **babyCaret** has over **arules**. Tutorial users are able to change the minimum support for an itemset to be considered frequent. This is done through the `minSup` parameter. This sets the frequency required for an itemset to avoid pruning. Reducing `minsup` reduces the number of infrequent itemsets found and therefore reduces runtime. The minimum confidence for a rule to be displayed can be controlled by the `minConf` parameter. This sets the minimum confidence a rule must have to be returned to the user. The maximum length of an association rule is controlled by the `maxLen` parameter. This sets the maximum allowed iterations of the process generating candidate itemsets. Since candidate itemset length or size grows by one each iteration and the association rules are created from the itemsets, controlling the maximum iterations of this process effectively controls the maximum length a rule may have; the length of a rule is simply the number of items composing the rule. As a whole, these parameters are used to reduce computation time and/or reduce the number and complexity of rules returned to the user. The description of the apriori algorithm given in **babyCaret's** tutorials can be found in Appendix A.8.

## 3.3 Tutorials

**babyCaret's** tutorials rely heavily on metaphorical explanations and user experimentation. The general template of a tutorial begins with text explaining the algorithm or problem interspersed with true/false and multiple choice questions. This is followed by programming questions which, for tutorials focused on algorithms, guide the users through training a

The image shows a screenshot of the RStudio application window. The title bar reads "RStudio". Below the title bar is a toolbar with various icons. The main window is divided into two panes: "Source" and "Console". The "Console" pane is active and shows the following text:

```
> tutorial()

      babyCaret tutorial menu
-----

1 --- What is babyCaret?
2 --- How to use these tutorials
3 --- Prerequisite R Programming
4 --- Creating Training and Testing Sets
5 --- Using the algorithms (in general)
6 --- K-nearest neighbors algorithm
7 --- K-prototypes algorithm
8 --- Apriori algorithm
9 --- Decision tree algorithm

Enter a number:
4|
```

Figure 3.5: **babyCaret**'s Tutorial Menu

model, evaluating the quality of the model, manually optimizing the model, and then using the model for prediction or data analysis. Tutorial content can be found in Appendix A.

The tutorials run entirely inside the R console. To enter the tutorial menu, the user runs the function `tutorial()`. As shown in Figure 3.5, this prints nine module options to the console for the user to select from. To select a module, they are prompted to select a corresponding integer. Once the user selects a menu option, **babyCaret** takes control of their console. The only action a user can make is to enter a character string which gets assigned to an R object. This state ends either upon completion of a module or the entry of one of **babyCaret**'s exit commands, which are "skip", "exit", "quit", and "bye". Multiple exit commands were chosen to avoid the user being required to memorize a specific command. Entering an exit command at any time will exit the tutorial and return full control of the

console to the user.

Tutorials consist of information displayed as descriptive text, multiple choice questions, true/false questions, and programming questions as shown in Figures 3.6, 3.7, and 3.8. This thesis refers to the display of any of these discrete elements as a frame. The text information is displayed by a C++ function wrapped by a private R function `.linePrint()`. Note, all functions beginning with `'.'` are private R functions created during development. This developer-defined function was preferred because when text reaches the end of a line using R's built in `print()` or `cat()` functions, any characters that do not fit on that line are displayed on the next line, even if it was in the middle of a word. This can increase the difficulty of reading displayed text. The extra control C++ gives over character strings allowed for creation of a simple function that requires the entirety of a word to be printed on the same line. This was done by setting a fixed maximum line length. Fixing line length had the trade-off of not allowing the line length to respond to the user increasing the size of their console window. The default behavior can still take over when decreasing the window below the set line length. At the end of a section of text, the user is prompted to press enter to continue. When they do, this moves them to the next frame. Alternatively, they may use an exit command to leave the module.

Multiple choice questions use **babyCaret's** `.multipleChoice()` function. This prints a question followed by four answers (Figure 3.6). Each answer is tagged either "a", "b", "c", or "d". The arrangement of the answers is randomized. The user enters a letter to select an answer. If they get the incorrect answer four times, the question is re-displayed. To skip any question, they may enter one of three skip commands: "skip", "next", or "ugh". These skip commands work on any question type. Once the correct answer is given or the question is skipped, the user is moved to the next frame.

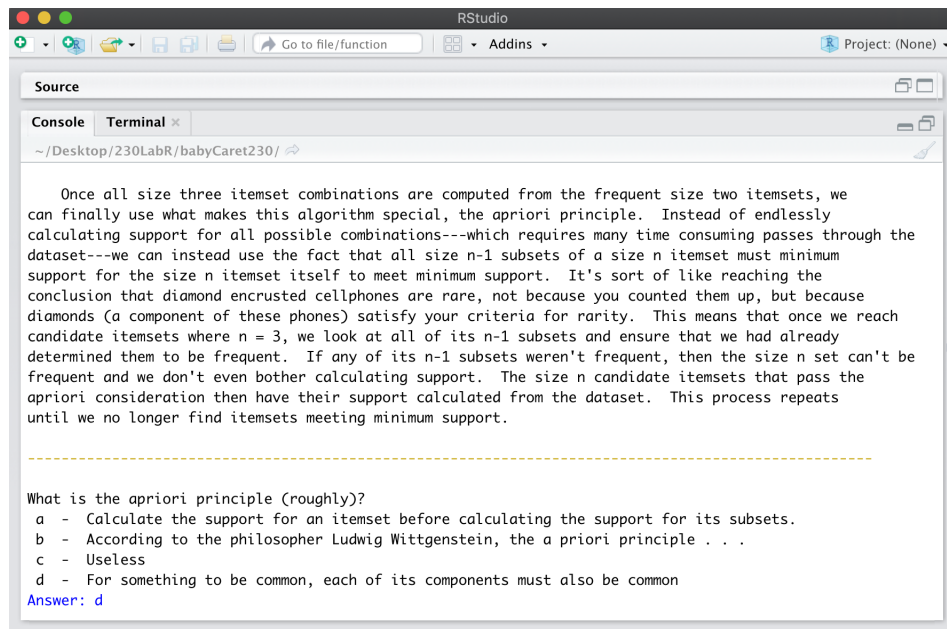


Figure 3.6: Multiple Choice Question about the Apriori Algorithm

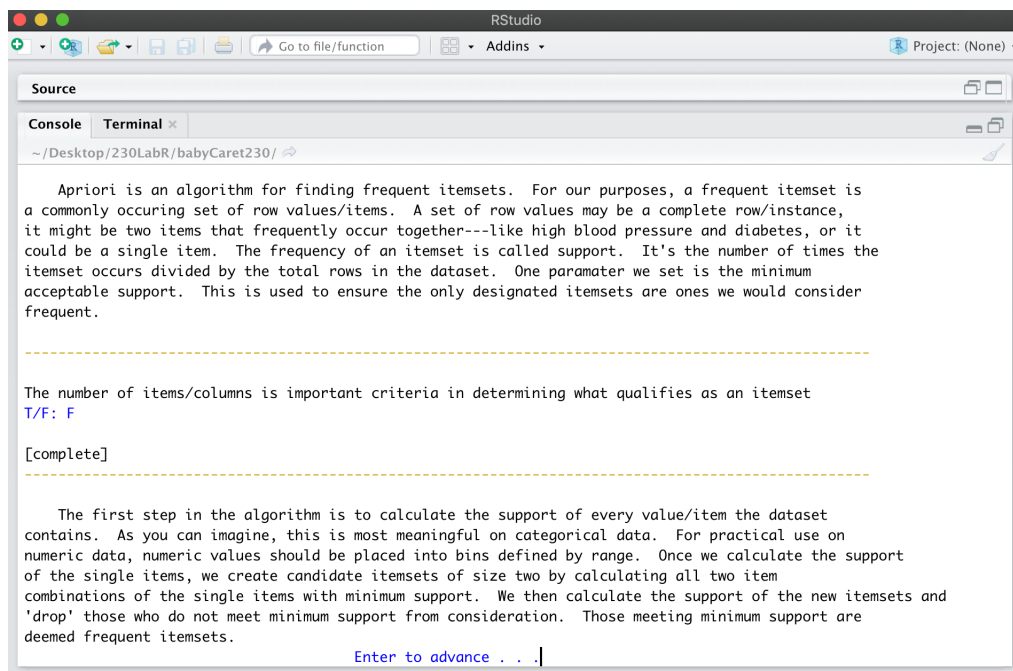


Figure 3.7: True/False Question about the Apriori Algorithm

```

RStudio
Go to file/function
Addins
Project: (None)

Source

Console Terminal x
~/Desktop/230LabR/babyCaret230/

-----

Train k-prototypes on the iris dataset with k = 4. You don't need to assign the result to a variable or include any
other parameters
babyTrain(NULL, iris, "kproto", k = 4)
Numeric predictors: 4
Categorical predictors: 1
Lambda: 1

Number of Clusters: 4
Cluster sizes: 28 50 40 32
Within cluster error: 14.88958 24.08526 22.23344 22.39944

Cluster prototypes:
  Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
1    5.532143    2.635714    3.960714    1.228571 versicolor
2    5.006000    3.428000    1.462000    0.246000  setosa
3    6.252500    2.855000    4.815000    1.625000 versicolor
4    6.912500    3.100000    5.846875    2.131250  virginica

[complete]

-----

Within cluster error is a measure of how well the prototype represents the cluster. When k
remains constant, a lower within cluster error is preferred.
Enter to advance . . . |

```

Figure 3.8: Programming Question Related to the  $k$ -prototypes Algorithm

True/false questions use **babyCaret's** `.trueFalse()` function. The question is displayed followed by T/F (Figure 3.7). The user may enter their response in forms such as “T”, “TRUE”, “f”, or “false”; case does not matter. Once the correct answer or a skip command is entered, the user is moved to the next frame.

Programming questions prompt the user to enter code to achieve a certain task (Figure 3.8). These tasks include standard use of R and use of **babyCaret's** ML functionality. Evaluation for the correctness of code is straight forward if only one answer is considered correct. However, for most (if not all) programming questions, there are many correct answers. At a high level, there are different ways to go about solving a compound problem. The issue of multiple approaches is dealt with by walking step-by-step through compound problems with the user. An example of a compound problem would be extracting a subset of values from a dataset and then finding the mean. Additionally, this can foster learning;

guiding users step by step through a single approach allows them to mentally map out the problem space. They can then use what they have learned for independent exploration.

There are many possible correct responses to even non-compound programming questions. As a simple example, when asked to assign the string “hello” to an object named `x`, both `x <- "hello"` and `x <- "hello"` are correct. The difference between the two is trivial whitespace. However, when testing for equality between the two lines of code as character strings, they are not equal. This specific issue can be solved by removing all white space and then evaluating for equality. However, `x = "hello"` is also correct. The approach to overcome this was to evaluate the result of executing their code.

The majority of programming questions are handled by the `.askUntil()` function. One parameter of this function is `desiredInput` which accepts a line of code as its argument. The argument is evaluated and the result is assigned to an object. This is matched to the result of the desired user input. The user’s input is passed to the function as a string which is cast to an expression and then silently evaluated. If the evaluation does not cause an error, the result is assigned to an object. This process repeats, including receiving new user input, until the user’s result matches the desired result. Since the user has no control over the R console, they are not able to run code directly. This is handled by **babyCaret**. Once the the results are identical, `.askUntil()` will either assign the result to the global environment and/or print the output to the console for the user to see. This approach keeps the user sand-boxed and avoids them crashing the tutorial by entering error causing code.

There is one edge case where there are two correct results. This is dealt with by using an optional second parameter, `alternativeInput`. If used, the user’s result is compared to both results. There are two edge cases that were not working that involved the **rpart** package that **babyCaret** uses for decision tree computation. The objects would not evaluate



as identical, but this was solved by checking the **rpart** objects for equality of user specified parameters.

One last case led to problems in our approach: randomness. The issues were caused by partitioning a dataset or training  $k$ -prototypes. Partitioning data using **babyCaret's** `dataPartition()` function (shown in Appendix C) uses random sampling to split a dataset into training and testing sets.  $k$ -prototypes must initialize  $k$  random prototypes. To resolve the issues, a hidden random seed is added to the user's global environment before any programming question involving these two processes is asked. If either function detects the random seed, the function will use it. Once a question relying on a random seed is answered or skipped, the hidden seed is removed. If the user ever enters an exit command, the hidden seed is searched for and if present, removed before exiting. In addition to allowing for evaluation of correct user response, using a specified random seed for these functions in the tutorial environment allows for consistent output which can then be reliably used by the tutorials for discussion.

## Chapter 4

### Results

This chapter consists of results from the distribution of **babyCaret**'s tutorials as a classroom tool, followed by runtime analysis of **babyCaret**'s KNN and  $k$ -prototypes implementations. KNN and  $k$ -prototypes are included because the KNN implementation is unique to **babyCaret** and the  $k$ -prototypes implementation modifies Gero Szepannek's implementation [27]. Decision tree and apriori are not included in the result section because **rpart**'s decision tree implementation [28] and **arules**' apriori implementation [10] are used by **babyCaret** as package dependencies in unaltered form.

#### 4.1 Tutorial Distribution

**babyCaret** was distributed to a 200-level intelligent systems class as a part of the course's homework. This was done because **babyCaret**'s tutorials aligned with course goals. Basic feedback on the users' experiences was given to improve the tutorials and instructions for using the tutorials. IRB approval for user testing was pending when face to face interactions were interrupted by COVID-19 responses and time was limited for implementation and evaluation.

Table 4.1: Mean KNN runtime in milliseconds on Fisher’s Iris dataset with  $k = 5$ 

KNN version	Training (ms)	Prediction (ms)	Total (ms)
<b>caret</b>	559.9	2.38	562.3
<b>babyCaret</b>	0.20	12.38	12.58

## 4.2 $k$ -nearest Neighbors Runtime

**babyCaret’s**  $k$ -nearest neighbors implementation was applied to Fisher’s Iris dataset [8] and compared to the implementation used in **caret**. The two implementations predicted values from the same dataset. Training was performed on 113 instances and prediction on the remaining 37 instances. The argument for KNN’s single parameter,  $k$ , was five in both implementations. Both training and prediction were performed 100 times. Mean runtime is shown in Table 4.1. **babyCaret’s** mean combined runtime for training and prediction is 549.72ms ( $44.70\times$ ) faster than **caret’s**. **babyCaret’s** prediction process is slower by approximately 10.00ms ( $5.21\times$ ). This was expected because even without performing cross validation, **caret’s** training process has more overhead than **babyCaret’s** does. This overhead is shown by **caret’s** model containing 124,272 bytes when trained on the entirety of the iris dataset in comparison to **babyCaret’s** 8,272 bytes. **caret’s** model includes the function call, a vector of attributes, a list of user specified and default training options, and more. **caret’s** list of options alone contains 25,168 bytes. Even without using advanced functionalities such as cross validation, their existence still causes overhead because memory must be allocated within this list of options to tell **caret** *not* to do that processing. **babyCaret’s** model contains only the minimum information needed to make predictions.

Table 4.2: Mean  $k$ -prototypes runtime in milliseconds on Fisher’s Iris dataset with  $k = 5$ , `nstart = 1`, `iter.max = 100`.

$k$ -prototypes version	Training Time (ms)
Original	79.93
Modified	39.24

### 4.3 $k$ -prototypes Runtime

Gero Szepannek’s  $k$ -prototypes R implementation and **babyCaret**’s modified version of Szepannek’s implementation using C++ were trained on Fisher’s Iris dataset. All hyperparameters were set to identical values. Both implementations ran inside of **babyCaret** to control for package overhead. Training for each implementation was performed 100 times. Prediction runtime was not measured because the modifications made were unique to the training process. Mean runtime is shown in Table 4.2. The modified implementation’s mean runtime was faster by approximately 40ms (2.04 $\times$ ). This was expected because the use of R throughout the entirety of Szepannek’s implementation causes increased lookup of methods and object initialization.

## Chapter 5

### Conclusions & Future Work

The machine learning tutorials were successfully implemented in R. The tutorials worked as intended by delivering text information and multiple forms of questions without the user having to leave their programming environment. They were installed and used to supplement an intelligent systems class, but without formal user feedback, it is difficult to discuss their usefulness and functionality. In this chapter, implementation performance is discussed followed by future work. Future work can include user evaluation, moving tutorials to the **learnr** platform, and modifying implementations.

#### 5.1 Assessing Implementation Performance

In total runtime, **babyCaret**'s KNN implementation was shown to be faster than **caret**'s by a factor of 44.7. This is useful for progressing through tutorials without delay and shows the time to implement KNN using C++ was well spent. Although it is difficult to be certain, the primary cause of the speed-up is suspected to be that the implementation is designed to have the minimum functionality required to make predictions and support **babyCaret**'s tutorials. Assuming the use of C++ alongside R did lead to some decrease in runtime, it is not likely to be the primary factor responsible for the decrease.

The training time for **babyCaret**'s KNN implementation was significantly faster than

**caret**'s KNN implementation. This is indicative of **babyCaret**'s simpler training process. When one of the main uses of the implementation is as a teaching topic for novice users, simpler should be preferred.

**babyCaret**'s implementation is slower than **caret**'s at KNN prediction. One reason is that **babyCaret** scales its training data during the prediction process, whereas **caret**'s training data is scaled during the training process, which increases **caret**'s training time. This has no effect on the predicted values since it is still done before calculating distances.

Using C++ to calculate the distance matrix for **babyCaret**'s  $k$ -prototypes implementation showed a decrease in runtime of approximately 50%. This validates our approach of using a combination of C++ and R as opposed to only using R.

The decision tree and apriori algorithms functioned well within **babyCaret**'s tutorials. This was expected because no modifications were made to either implementation.

## 5.2 Future work

There are multiple areas that could be addressed to improve and further develop **babyCaret**. These include doing formal user testing of the tutorials, working in a new release of RStudio, and improving algorithm implementations.

### 5.2.1 User Evaluation

Formal feedback can be collected in the future during structured evaluation sessions. Materials for gathering this feedback to support both evaluation and improvement of the tutorials can be found in Appendix B.

### 5.2.2 RStudio 1.3

A new release of RStudio is currently under development, RStudio 1.3 [1]. One of its features will be an integrated tutorial window, which will be able to host tutorials made for RStudio's **learnr** platform [26]. Developers will be able to create tutorials for their R package and link to them from the package. The main focus of future work will be porting **babyCaret's** tutorials to the **learnr** platform and improving them using the tools provided by the platform. This will allow expansion of the features in **babyCaret's** tutorials to include interactive graphs, videos, images, and equations. RStudio's tutorial window will run completely independent of the user's R session, allowing users to use their R console for experimentation without having to leave the tutorial. Much of the work done on **babyCaret's** tutorials will be usable in the new format. **learnr** has modules that evaluate the correctness of programming question answers, so ideally **babyCaret's** answer checking functionalities should be usable after some modification [29].

### 5.2.3 Cross Validation

Support for cross validation should be implemented in the future. **caret** uses cross validation to test multiple hyperparameter configurations to automatically choose the most accurate configuration. Since this is so common in ML, it is a high priority feature. Automatic hyperparameter selection goes against **babyCaret's** purpose as an educational and experimental tool. However, cross validation in **babyCaret** would likely be a dynamic process that allows users to easily and manually tune hyperparameters and react to how they affect model accuracy.

## 5.2.4 KNN

Experimentation with a lambda parameter similar to the one found in Szepannek’s  $k$ -prototypes could be done. This would increase or decrease the importance of categorical attributes. Additionally, scaling the training data should be moved outside of the prediction process and into the training process.

## 5.2.5 $k$ -prototypes

In addition to the completed work, Szepannek [27] suggested future work should focus on automated selection of the lambda parameter. Although automated parameter selection is not in the development plans for **babyCaret**, this still should be considered to improve Szepannek’s implementation for users outside of **babyCaret**.

## 5.2.6 Apriori

Two attempts at an apriori implementation never made it into **babyCaret**. One of these was written in C++. This was abandoned mostly due to time and the need for major modifications to increase efficiency. Completion is a primary goal of future work.

Another attempt was written in R. The purpose of this implementation is to be used as a prototype for the C++ version. This implementation is currently more complete than the C++ version and is capable of finding frequent itemsets of size two and generating candidate itemsets of size three. Remaining tasks include checking the size  $N - 1$  candidate subsets against the size  $N - 1$  frequent itemsets, pruning candidates using that information, calculating the support of the remaining candidates, pruning those not meeting minimum support, and then looping that process.



Development of our R version of the apriori algorithm should be completed and development should then return to the C++ version. Once apriori finds frequent itemsets, another algorithm is required to find association rules from those itemsets. In common usage, the apriori algorithm is considered the total process of finding itemsets and then rules, since after apriori is used to find itemsets, the immediate next step in data mining is typically to use another algorithm for finding association rules. Technically, apriori is just the algorithm that finds frequent itemsets and our R version only finds frequent itemsets with a maximum length of two items. Significant further development is required to implement the full process of finding both frequent itemsets and their association rules. This extension was beyond the scope of this thesis, but is valuable future work.

# Bibliography

- [1] RStudio v1.3.957-1 preview, 2020. <https://rstudio.com/products/rstudio/download/preview/>.
- [2] M. Akamine and J. Ajmera. Decision tree-based acoustic models for speech recognition. *EURASIP Journal on Audio, Speech, and Music Processing*, 10, 2012.
- [3] R. Argawal and R. Srikant. Fast algorithms for mining association rules. *Proceedings of the 20th VLDB Conference*, 1994.
- [4] L. Brieman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.
- [5] R. Donovan. Topics in decision tree based speech synthesis. *Computer Speech Language*, 17(1):43 – 67, 2003.
- [6] D. Eddelbuettel and J. J. Balamuta. Extending R with C++: a brief introduction to Rcpp. *The American Statistician*, 72(1):28–36, 2018.
- [7] D. Eddelbuettel and R. François. Rcpp: Seamless R and C++ integration.
- [8] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

- [9] H. Golino and C. Gomes. Random forest as an imputation method for education and psychology research: its impact on item fit and difficulty of the Rasch model. *International Journal of Research & Method in Education*, 39(4):401–421, 2016.
- [10] M. Hahsler, S. Chelluboina, K. Hornik, and C. Buchta. The arules R-package ecosystem: Analyzing interesting patterns from large transaction datasets. *Journal of Machine Learning Research*, 12:1977–1981, 2011.
- [11] W. Hu, X. Li, T. Wang, and S. Zheng. Association mining of mutated cancer genes in different clinical stages across 11 cancer types. *Oncotarget*, 7(42):68270–68277, 2016.
- [12] Z. Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3):283–304, 1998.
- [13] R. Ihaka. A free software project, 2017. [https://cran.r-project.org/doc/html/interface98-paper/paper\\_2.html](https://cran.r-project.org/doc/html/interface98-paper/paper_2.html).
- [14] S. Kross, N. Carchedi, B. Bauer, and G. Grdina. *swirl: Learn R, in R*, 2019. R package version 2.4.4.
- [15] M. Kuhn. Building predictive models in R using the caret package. *Journal of Statistical Software, Articles*, 28(5):1–26, 2008.
- [16] M. Kuhn. The caret package, 2020. <http://topepo.github.io/caret/available-models.html>, Accessed: 2020-05-4.
- [17] R. I. Lerman and S. Yitzhaki. A note on the calculation and interpretation of the Gini index. *Economics Letters*, 15(3-4):363–368, 1984.
- [18] U. Ligges and J. Fox. How can I avoid this loop or make it faster? *R News*,.

- [19] S. Milborrow. *rpart.plot: Plot 'rpart' Models: An Enhanced Version of 'plot.rpart'*, 2019. R package version 3.0.8.
- [20] V. Prasath, H. A. A. Alfeilat, O. Lasassmeh, A. Hassanat, and A. S. Tarawneh. Distance and similarity measures effect on the performance of k-nearest neighbor classifier—a review. *arXiv preprint arXiv:1708.04321*, 2017.
- [21] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019.
- [22] K. Rantil, K. Salim, and A. Girsang. Clustering STEAM user behavior data using k-prototypes algorithm. *Journal of Physics: Conference Series*, vol. 1367(012018):1–7, 2019.
- [23] RStudio Team. *RStudio Cloud*. RStudio, Inc., Boston, MA.
- [24] RStudio Team. *shinyapps.io by RStudio*. RStudio, Inc., Boston, MA.
- [25] RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA, 2015.
- [26] B. Schloerke, J. Allaire, and B. Borges. *learnr: Interactive Tutorials for R*, 2019. R package version 0.10.0.
- [27] G. Szepannek. clustMixType: User-friendly clustering of mixed-type data in R. *The R Journal*, 10(2):200–208, 2018.
- [28] T. Therneau and B. Atkinson. *rpart: Recursive Partitioning and Regression Trees*, 2019. R package version 4.1-15.

- [29] K. Ushey. RStudio 1.3 preview: Integrated tutorials, 2020. <https://blog.rstudio.com/2020/02/25/rstudio-1-3-integrated-tutorials/>, Accessed: 2020-05-6.
- [30] H. Wickham. *Advanced R*, chapter Performance, pages 331 – 334. Taylor & Francis, 2014.
- [31] H. Wickham. *Advanced R*, chapter High performance functions with Rcpp, pages 395 – 401. Taylor & Francis, 2014.
- [32] G. N. Wilkinson and C. E. Rogers. Symbolic description of factorial models for analysis of variance. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 22(3):392–399, 1973.
- [33] D. Şengür and M. Turhan. Prediction of the action identification levels of teachers based on organizational commitment and job satisfaction by using k-nearest neighbors method. *Turkish Journal of Science Technology*, 13(2):61–68, 2017.

## Appendix A

### babyCaret's tutorials

This appendix shows shows the text descriptions and questions from each tutorial formatted for this appendix.

#### A.1 What is babyCaret?

What is babyCaret?

---

babyCaret is a simple machine learning (ML) package for the programming language R. It is one part entry-level ML package and one part teaching tool. In an attempt to be approachable by new users, babyCaret includes tutorials on both its use and ML in general. These tutorials run in an R using babyCaret's own tutorial engine ("engine" might be a bit of a stretch) and are intended to be used within the RStudio IDE. In addition, their development is an experiment on the viability of console based tutorials as an alternative form of package documentation. babyCaret has syntax inspired by R's primary ML package, Caret, and aims to ease the user's transition into Caret.

#### A.2 How to Operate babyCaret's Tutorials

How to Operate babyCaret's tutorials

---

The tutorials will run entirely in the console (bottom left window). To enter the tutorial menu, type `tutorial()` in the console and then press enter. `tutorial()` does not force you to make a selection from the menu; if you make an invalid selection, you must use `tutorial()` again. Currently, `tutorial()` must also be used to manually navigate to the next module upon completion of a module. To exit a tutorial at any time, type `stop`, `quit`, `bye`, or `exit` into the console, then press enter. IF YOU ENCOUNTER A BUG OR GET STUCK ON A QUESTION, type `skip`, `next`, or `ugh`, then press enter to skip that question. If the tutorial crashes, use `tutorial()` to re-enter the module and use one of the skip commands to navigate to return to the spot it crashed at. It's recommended but not required to clear your workspace before beginning a tutorial. To do so from RStudio, click the broom in the upper right window. Don't be afraid to scroll up through the console if you want to reference previous answers or information. The rest should be fairly self explanatory. Check back for more detailed instructions in a future version.

## A.3 Prerequisite R Programming

Prerequisite R programming

---

First, lets go over a couple of terms.

**Object:** An object in R is anything that has a value assigned to it. The value could be a number, word, or even an entire dataset. It's essentially a variable.

**Operator:** An operator performs an operation. This could be addition, a logical operation (like OR or AND), the assignment of a value to an object, or something else entirely.

---

One of the operators used most frequently is the assignment operator. This is used to assign a value to an object. The assignment operator is a less-than symbol followed by a minus symbol. Basically, it's a left pointing arrow (`<-`). The name of the object you are assigning a value to is placed on the left side of the arrow. The value you are assigning that object is placed on the right side. You can think of the arrow as shoving a value into an object.

---

Here's an example: `yourObject <- 22`

This line of code assigns the value 22 to a new object named `yourObject`. `yourObject` is a new object because this is the first time it has had a value assigned to it. `yourObject` can now be used as a stand-in for 22.

---

```
myObject <- . . . Assign the value "hello world" to a new object named
myObject:
```

```
skip
```

```
[complete]
```

---

If you're wondering what the `"myObject <- . . ."` is for, it's to inform that you are supposed to assign something to an object called `myObject`. The assignment request of a coding prompt can easily fall out of your brain, so you'll see that format a lot in these tutorials.

---

If you want to see (or show me) the value of `myObject`, all you have to type is its name. You can also see it in the upper right window, the variable explorer.

---

```
Show me the value of myObject:
```

```
myObject
```

```
[1] "hello world"
```

```
[complete]
```

---

We call doing what you just did, "printing `myObject` to the console". Doing so allows you to see the value of `myObject` in the window you're currently interacting with, the console.

---

If you're planning on crunching data through a machine learning algorithm, you have to be familiar with data frames. In R, data frame objects have the 'type' `data.frame`, so I'll often refer to them as such. A `data.frame` is R's representation of a spreadsheet. Each row of a `data.frame` contains a single



instance and each column is an attribute those instances have. If we were using data on people, each instance (row) would be a specific person and each attribute (column) would be a measurement or quality of those people. These attributes may be things like weight, blood pressure, marital status, etc.

---

I'm going to add a data.frame called cars into your R session. You should see its name appear in your variable explorer (upper right). To display the first six instances, enter the command: `head(cars)`. This command is useful for determining the attributes of a data.frame without being overwhelmed by every single instance it contains.

---

Display the first six instances of the cars data.frame

```
skip
speed dist
1 4 2
2 4 10
3 7 4
4 7 22
5 8 16
6 9 10
[complete]
```

---

From this, we can see that the first car in this dataset has a speed of 4 and a dist of 2. This data.frame only has 50 rows, so it wouldn't have been a big deal to print its entirety to the console, but if it contained thousands of rows, seeing the whole thing would be a bit of a distraction if you just wanted to see its attributes and a few example instances. If we did that, you would have to do a lot of scrolling up just to see the attributes displayed at the top.

---

Sometimes you'll want to do something with a specific column (attribute) of a data.frame. You can access a column with the `$` operator. When you type the data.frame's name followed by `$` and then the column's name, you will

grab only the specified column from that data.frame. You can then use that column for whatever you want---maybe as target data for a machine learning algorithm?

---

Here's an example: `myDf$eyeColor`

This line of code accesses `eyeColor` column of a data.frame named `myDf`. If you wanted, you could even assign this data to a whole new object. One reason to do so would be to manipulate a copy of that data without modifying the source.

---

`vroom <- . . .` Assign the speed column of the cars data.frame to a new object called `vroom`:

```
vroom <- cars$speed
```

```
[complete]
```

---

Now print the contents of `vroom` to the console

```
skip
```

```
[1] 4 4 7 7 8 9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14 15 15 15  
16 16 17 17 [31] 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24 24 24 24 25
```

```
[complete]
```

---

Technically, `vroom` is a vector. A vector is an ordered collection of values.

---

We have one last topic to cover before moving on: functions. A function is a sequence of operations performed by R. The nice thing is that you've already used one function. When you used `head(cars)`, that was a function. R has many built in functions. When we use a function, we refer to it as a function 'call'. What you did was call the `head()` function on the cars data.frame. One way to think about it is that you headed cars. In this context, we refer to cars as an 'argument'. An argument is anything that goes inside of a function's parentheses. The function uses those arguments to carry out its processes. Different arguments can have different roles. An argument may be data to be processed or a value specifying how to process

data. The value that comes out of a function once it is finished with its processes is said to be 'returned' by the function.

---

Use a function to find the mean value in vroom skip [1] 15.4

[complete]

---

We're going to use the mean function to learn a little more about functions in general. Don't worry too much about the particularities of the mean function.

---

Function call: `mean(x, trim = 0, na.rm = FALSE)`

Parameters: --- x --- An R object

--- trim --- the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed.

--- na.rm --- a logical value indicating whether NA values should be stripped before the computation proceeds.

---

It's worth taking a little bit of time to get familiar with the form the mean function is in presented above. We will use this form frequently. The "Function call:" section shows how to type out a call to this function. x, trim, and na.rm are referred to as parameters. When you're actually typing out this function call, instead of typing 'x', 'trim', or 'na.rm' you will enter the arguments you want to stand in for x, trim, and na.rm. The arguments can either be objects containing a value you wish to use or explicit values. If you see an = in the call, it means the following value will be used if you do not enter a value; it's a default value. If you wish to use a default value, just don't include an argument for it.

---

The 'Parameters:' section describes how the arguments you enter to stand in for these parameters will be used by the function. When including arguments in a function call, you can either enter your arguments in the same order as presented in the function call section or explicitly say parameter = argument (trim = .25). Arguments must be separated by commas.

---

```
Call mean using vroom for x, .5 for trim, and FALSE for na.rm
```

```
skip
```

```
[1] 15
```

```
[complete]
```

-----

This is the end of the Prerequisite R Programming module. I'll go ahead and clean my garbage out of your variable explorer/R session. There is a lot more R programming to learn, but this should be enough to move you through the rest of the modules. If you would like to learn more R programming, you should try the R package Swirl. It was part of the inspiration for babyCaret!

## A.4 Creating Training and Testing Sets

Creating Training and Testing Sets

-----

For a couple of the algorithms (k-nearest neighbors and decision tree) included with babyCaret, you should consider having two separate datasets available. One of them is called a training set and the other is called a testing set. The training set is used in the learning process to create a model, and the testing set is used to evaluate how well the model performs.

-----

A training set is like study material for the algorithm. The algorithm studies and creates a model; much like how after you study chapter 2 section 6 of "Biology" by Raven et. al, the result is a mental model of how mitochondria work. If someone wants to determine the quality of your mental model, they would test you on relevant material you've never seen, but which they know perfectly. That is essentially what we are doing with the testing set.

-----

Mitochondria is:

- a - Where the transcription of DNA to RNA takes place
- b - The powerhouse of the cell

c - A neurotransmitter

d - A virus

Answer: b

[complete]

---

You might have both of these datasets provided to you, but often you will find yourself with only one large dataset. In this situation, you will need to split the dataset into a training and a testing set. A naive approach would be to chop the dataset in half, with the first 50% of instances going to the training set and the last 50% of instances going to the testing set. There are (at least) two problems with this. The first is that we aren't assigning randomly from the original dataset. This can have the unfortunate effect of causing our training set to be unrepresentative of the data. If the original dataset was created by entering a new instance every day from January 1st until December 31st, our algorithm would never be exposed to a single fall instance. That's not good; there are likely valuable things to learn from those fall instances that can't be learned from only winter and spring instances. The second problem with our naive approach is that we have too many instances in the testing set. Metaphorically, we want our algorithm to study a book and then take an intense and lengthy exam, not study half of a book and then take a test on the other half. A testing set containing between 20% and 30% of the original data is standard.

---

To partition a dataset into a training and a testing set, we are going to use `caret`'s `partitionData()` function. `partitionData` treats the original dataset like a deck of cards. First it shuffles the instances (cards), then it cuts the shuffled dataset (deck) into two separate datasets. The ratio of the cut is determined by you.

---

When using `partitionData()` you will type inside of the parentheses the name of the dataset followed by a comma, then the ratio of data you want in your training set as a decimal value. If you do not specify a ratio, `.75` will be used.

---

Here's an example: `partitionData(psychData, .7)`

In the above call to the function `partitionData()`, we are partitioning a

dataset called `psychData` into a training set containing 70% of the original data and a testing set containing 30% of the original data. The next frame will show a more formal explanation of this function.

---

```
Function call: dataPartition(df, p = .75)
```

```
Parameters:
```

```
--- df --- The dataframe object you are partitioning
```

```
--- p --- A number between 0 and 1. This determines how large your testing set is. If .7 is entered, 70% of the original data will be in the testing set and 30% will be in the training set. If no value is entered, p will = .75
```

---

```
Call dataPartition() using iris as df and .8 as p: dataPartition(iris, .8)
```

```
[complete]
```

---

```
Just calling dataPartition() isn't enough for our purposes.
```

---

```
myData <- . . . Repeat the same function call as before, but assign it to a data object named 'myData':
```

```
skip
```

```
[complete]
```

---

```
You may have noticed that we have only added one variable to the variable explorer 'myData', but are trying to create two new datasets. The training and testing sets are both contained in the myData object. We can access them by using the $ operator. It's just like accessing a column from a data.frame. myData$train will return the training set, while myData$test will return the testing set.
```

---

```
trainSet <- . . . Assign the training set to a data object named trainSet
```

```
skip
```

```
[complete]
```

---

```
testSet <- . . . Assign the testing set to a data object named testSet
testSet <- myData$test
[complete]
```

---

Your training and testing sets are now ready ready for use! I'll go ahead and remove myData from your variable explorer

---

Well, I guess I'll remove everything because you've reached the end of the Creating Training and Testing Sets module. We've only covered the basics of training and testing sets, but it's enough to get us pretty far. Enter to advance . . .

## A.5 Using the Algorithms (in general)

Using the Algorithms (in general)

---

When training any model in babyCaret, the babyTrain() function is used.

---

Function call: babyTrain(formOrY, data, method, ...)

Parameters: --- formOrY --- There are two options: a formula or the target attribute as a vector

--- data --- the training set as a data.frame object

--- method --- the algorithm you wish to use

--- ... --- parameters specific to each algorithm

---

formOrY is short for 'formula or y'. This should give you the clue that there are two formats you can use for this argument. The first is formula notation. In babyCaret, formula notation can be used to specify which attributes in a data.frame you think predict your target variable/attribute---what you're trying to predict. This is useful when

you don't think all attributes in a dataset are useful in predicting your target attribute. Sometimes less is more.

---

Here's an example: `myData$sleep ~ myData$programming + myData$coffee`

This formula can be read as, "sleep is predicted by programming and coffee". Each chunk of data must be a vector you subset from the training set. The target attribute is the left most entry in the formula. This is followed by a tilde (~), which basically means, "is predicted by". After the tilde, you type the attributes you think predict the target attribute separated by pluses (+). You aren't limited to just two predictors like the example. You could make a formula with 8 predictors if you wanted.

---

Data used in a formula must be vectors subsetted from the training set.

T/F: T

[complete]

---

If you don't use a formula, you'll just enter the target attribute as a vector subsetted from the training set. To do so, `formOrY`'s argument be something like `myData$sleep`. This means you'll be building a model which uses every attribute in the dataset for prediction (which sometimes isn't best).

---

The `data` parameter should be provided with the entire training set.

---

The `method` parameter accepts one of four character (string) arguments, "knn", "kproto", "apriori", "tree", or "hmm". Quotes are required. "hmm" is used for tutorial purposes only.

---

The `...` parameter is a placeholder for method specific arguments. These must be entered after the first three arguments.

---

In order to use the model trained by `babyTrain()` for prediction, you must assign the result of the function call to an object.



---

I've put a data.frame called myTrainSet into your R session. Print it to the console.

```
skip
att targ
1 hmm uhm
2 hmm uhm
3 hmm uhm
4 hmm uhm
5 hmm uhm
[complete]
```

---

myModel <- . . . Train a "hmm" model which uses myTrainSet\$att to predict myTrainSet\$targ. For the value of hmm's model specific parameter 'aNumber' use 42. Assign the model to a new object called myModel

```
myModel <- babyTrain(myTrainSet$att, "myTrainSet$targ", "hmm", 42)
[complete]
```

---

Once you have a model, you can begin making predictions. Predictions are made using the babyPredict() function.

---

Function call: babyPredict(trainedModel, newdata, isTest = FALSE)

Parameters: --- trainedModel --- A model trained by babyTrain()

--- newdata --- The dataset with values to be predicted.

--- isTest --- A logical value indicating whether or not the prediction is being used on a testing set with known values. When set to true, returns information on model accuracy. Must be TRUE or FALSE. Defaults to FALSE.

---

The trainedModel parameter will be provided with a trained model like myModel (which you just trained). The newdata parameter requires a data.frame formatted EXACTLY like the training set. The values for the

target attribute can be missing or 'NA' when you're predicting unknown values. `babyPredict()` will fill in those missing values with predicted values. If the values aren't missing, `babyPredict` will rewrite the known values with its predicted values. The results can then be used for evaluating model performance outside of the tools for doing so provided by `babyCaret`. the `isTest` parameter is set to `FALSE` unless you're using a testing set with known values to evaluate model performance, in which case it should be `TRUE`.

---

I've put a `data.frame` called `unknownSet` into your R session. Print it to the console.

```
skip
att targ
1 hmm NA
2 hmm NA
3 hmm NA
4 hmm NA
5 hmm NA
[complete]
```

---

Use `babyPredict()` to predict the missing Values in `unknownSet`

```
babyPredict(myModel, unknownSet)
att targ
1 hmm ???
2 hmm ???
3 hmm ???
4 hmm ???
5 hmm ???
[complete]
```

---

As you can see, "hmm" is a very low quality machine learning algorithm.

??? obviously isn't the real value for targ, but atleast it predicted some values. It did the best it could. This is the end of the Using the Algorithms (In General) module. I'll go ahead and clean my junk out of your R session. Enter to advance . . .

## A.6 $k$ -nearest Neighbors algorithm

### K-Nearest Neighbors

---

K-nearest neighbors (KNN) is an algorithm that is used to predict unknown numeric or categorical values. This could be anything from species of flower, finishing position in a race, or the sale price of a house. Suppose you want to follow the KNN algorithm with pen and paper to predict the sale price of an interesting house in your neighborhood. Your first step would be to make a list of known sale prices (this is your target attribute), addresses, and square footage for other houses in the neighborhood. Your second step would be to make a sub-list consisting of the information belonging to the  $k$  ( $k$  is a whole number chosen by you) houses nearest in address and square footage to the one you're interested in. Lastly, you would take the  $k$  sale prices in your sub-list, add them all up, and divide by  $k$ . Your hope is that the average sale price of the  $k$  closest and most similar houses will be an accurate prediction.

---

The learning aspect of KNN is simple. All KNN has to do is store the entire dataset in memory. The specific implementation included with `babyCaret` also scales numeric data between 0 and 1. This is done to reduce the impact of differing units of measurement. What you need to specify is an integer value for  $k$ , and what attribute you want it to predict. The value selected for  $k$  will be set in the the model as how many of the closest neighbors to average when making a prediction. If  $k$  is set too high, you're likely to consider very distant and irrelevant values. It would be like considering the entire city instead of just the neighborhood. If you set  $k$  as high as it can reasonably go (the number of instances in the training set), then for every prediction it makes, KNN will just return the mean of the training set's target variable---the variable you're telling it to predict. If you set  $k$  too low, you reduce your buffer against anomalous neighbors throwing off your predictions. Imagine being told you're exactly the same as your

intolerable nextdoor (k = 1) neighbor.

---

Which technique is used in KNN's learning process?

- a - Storing the training set in memory
- b - Defining a set of rules
- c - Learning a polynomial
- d - Finding a hyperplane

Answer: a

[complete]

---

In order to make a prediction, KNN must calculate the distance between the instance being predicted and every instance in the training set. Let's focus on a single iteration of this process. babyCaret's KNN implementation primarily uses a distance metric called Manhattan distance. A precise-ish definition of Manhattan distance is the sum of the absolute values of the difference between corresponding dimensions. If that's a bit much, we can learn a lot about Manhattan distance by beginning with its name. Imagine you're in Manhattan and are trying to drive to a pizza place 2 blocks north and 1 block east. The distance to you, the driver, is 3 blocks. This is because you are restricted to only driving on roads. Distance "as the crow flies" would be Euclidean distance. If you arrived to discover that the pizza place was on the 4th floor of the building, the distance would then be 7 blocks because in our Manhattan, each floor has the height of one block. (0, 0, 0) was your starting point and your 'neighbor', the pizza place was at (2, 1, 4). Each position inside the parenthesis can be considered as an attribute or dimension. The Manhattan distance between these two locations is calculated as  $|0-2| + |0-1| + |0-4| = 7$ , with the vertical bars signifying absolute value.

---

Which operation is used when computing Manhattan distance?

- a - Appendectomy
- b - Square Root
- c - Exponentiation
- d - Absolute value

Answer: ugh

[complete]

---

If KNN is computing distance between ordinal attributes (must be ordered factor datatype, not character), then the values are computed as their underlying integer representation; on a scale of large, medium, and small, small would be converted to 1 while large would be converted to 3. These values are then used in the Manhattan distance calculation. If nominal attributes are being used (must be factor datatype, not character), then a match between two values has a distance of 1, else 0. This matching distance for nominal attributes is not Manhattan distance; it's an alternative metric used when Manhattan distance does not make sense. These values are added to the Manhattan distance.

---

How do you feel about one frame of optional (and potentially rambling) sidebar information?

1 - Good

2 - Not so good

2

After KNN has calculated the distance between the first test instance and the first training instance, KNN will then calculate the distance between the first test instance and every single training instance. Once complete, the average target value of the k nearest training instances is entered as the target value for the first training instance. If the target value is numeric or ordinal, this value is the average of target values from the k nearest training instances. If categorical, the mode is used rather than an average. Since for a categorical target variable, the mode is used, it makes sense to use an odd value for k. This prevents a 'tie', but only if the target variable is binary (two labels). Once the target value for the first test instance is entered, all that is left is to repeat this process for each test instance.

---

Before we move on to applying this algorithm, let me add a couple of things to to your R session. You should see them appear in the upper right window.

---

I've added some data on iris flowers to your R session. The iris dataset

is a classic, and if you continue your machine learning journey, you'll run into it a lot. `trainSet` is obviously your trainingSet, `testSet` is your testing set, `noPetalSet` is a dataset with unknown `Petal.Width` values, and `noSpeciesSet` has no `Species` values

---

Function call: `babyTrain(formOrY, data, method, ...)`

Parameters:

--- `formOrY` --- There are two options: a formula or the target attribute as a vector

--- `data` --- the training set as a `data.frame` object

--- `method` --- the algorithm you wish to use. "knn", "kproto", "tree", or "apriori". Make sure you include the quotation marks.

--- ... --- parameters specific to each algorithm

---

First, check out the missing values in `noPetalSet`. Print `noPetalSet` to the console. It's not very large.

`noPetalSet`

*This output omitted in appendix.*

[complete]

---

`myModel <- . . .` Use `babyTrain()` to train a KNN model which will use every attribute to predict the missing `Petal.Width` values. Use 75 as the value for `k`. Assign the result to an object named `myModel`

`myModel <- babyTrain(trainSet$Petal.Width, trainSet , "knn", 75 )`

[complete]

---

Before we predict missing values, let's use `babyPredict()` to find out how well our model performs on the testing set. This can be done by setting the `isTest` parameter to `TRUE`. First, I'll print some information on `babyTrain()` in case you need a refresher.

---

Function call: `babyPredict(trainedModel, newdata, isTest = FALSE)`

Parameters:

```
--- trainedModel --- A model trained by babyTrain()
--- newdata --- The dataset with values to be predicted.
--- isTest --- A logical value indicating whether or not the prediction is
being used on a testing set with known values. When set to true, returns
information on model accuracy. Must be TRUE or FALSE. Defaults to FALSE.
```

-----

Use babyPredict() to assess your model's performance on testSet. Don't assign the result to an object.

skip

```
[1] "Mean absolute percentage error: 64.9"
```

```
[complete]
```

-----

Mean absolute percentage error is a measurement of how accurate our numeric predictions are. It tells you, on average, how far away your predictions are from the the known values. We're using percentage error instead of raw error to account for the variation of magnitude in the true values. If we made two predictions 2 and 12 for the real values 4 and 14, our total absolute error would be  $2 + 2 = 4$  with both predictions contributing equally to the total error. However, the second prediciton is only off by about 17

-----

I think we can make this model more accurate. Let's try to reduce our error. We're likely considering too many neighbors and should reduce k. One thing to note. We're using performance metrics from our testing set to improve model accuracy. Ideally, we would have a special set for doing this and then use the testing set once to determine our models accuracy. Our metrics are likely to be less accurate in real world performance if we tune our model on the data that we are using to compute performance metrics. We can get into a situation where our model seems great but performs better on the testing set than all other datasets.

-----

```
myModel <- . . . Train a new KNN model to predict Petal.Width. Use k = 5
and assign the result to myModel.
```

skip

[complete]

---

Use `babyPredict()` to assess your model's performance on `testSet`. Don't assign the result to an object.

```
babyPredict(myModel, testSet, isTest = TRUE)
```

```
[1] "Mean absolute percentage error: 14.9"
```

[complete]

---

That looks better! let's use our model to predict the missing `Petal.Width` values in `noPetalSet`

---

```
newSet <- . . . Use babyPredict to predict the missing Petal.Width values. Assign the result to an object named newSet
```

```
skip
```

[complete]

---

Let's check out the predicted values. We can't claim with certainty how accurate the predictions are this time because we don't know the true values, but we can at least look at them and revel in your brilliance.

---

Print `newSet` to the console.

```
skip
```

*This output omitted in appendix.*

[complete]

---

Cool! If we were to look at the training set, you would be able to see that the `setosa` species usually has a pretty small petal width. Let's see if your predicted values make sense.

---

Print your training set to the console and then compare the petal width values from `setosa` plants to your predicted values above.



skip

*This output omitted in appendix.*

[complete]

---

Almost done. Before wrapping up, lets try predicting some categorical---nominal in this case---data.

---

```
myModel <- . . . Train a new KNN model which predicts trainSet$Species
from trainSet. Use k = 10 and assign the result to myModel.
```

skip

[complete]

---

Use babyPredict() to assess your model's performance on testSet. Don't assign the result to an object.

next

```
[1] "Percent misclassified: 3.84%"
```

[complete]

---

Easy as pie. We've reached the end of the KNN module. I'm going to leave everything in your R session this time. Feel free to use myModel to predict the values in noSpeciesSet!

---

## A.7 $k$ -prototypes algorithm

K-prototypes

---

K-prototypes is an algorithm used for clustering data. Its purpose is to group similar instances together. One reason to use a clustering algorithm is to apply labels to unlabeled data. Another is for exploratory data

analysis. If you have a bunch of data on similar dogs and notice that there are two distinct clusters, you now have a reason to investigate and see if you're actually dealing with two breeds.

---

First, we need to know what a prototype is. A prototype is our best representation of the typical member of a cluster. A cluster is a grouping of similar data points. Each cluster has exactly one prototype. A prototype doesn't have to be an actual member of the cluster. If you imagine the prototypical thanksgiving dinner, it's very possible that exact dinner has never existed. I'm pretty sure the Pilgrims never had access to Corn Flakes, but they're both a part of my prototypical thanksgiving dinner.

---

What is a prototype?

- a - The best representative of a cluster
- b - The squared variance of the mean
- c - An unfinished model
- d - A rough idea

Answer: d

Answer: a

[complete]

---

"Imagining" is not useful to the K-prototypes algorithm. To find a cluster's prototype, within-cluster averages (numeric) and modes (categorical) are calculated for each attribute. The prototype is then the instance (existing in the dataset or not) for which each attribute value is equal to its within cluster mean/mode.

---

The first step in the K-prototypes algorithm is to randomly choose k initial prototypes. These are chosen from real instances from the dataset.

---

After prototypes are chosen, each instance's distance from each prototype is calculated. Euclidean (straight line) distance is used for numeric attributes, for categorical variables, the same matching distance as K-nearest Neighbors is used. The total matching distance between a

prototype and an instance is simply added to the euclidean distance. If you wish, the total matching distance can be multiplied by a value, lambda, to either decrease or increase the importance of categorical variables.

---

What is the first step in the K-prototypes algorithm

- a - Separate numeric and categorical attributes
- b - Calculate distances
- c - Assign prototypes
- d - Randomly choose k initial prototypes

Answer: d

[complete]

---

After calculating distances, each instance is assigned to its closest/most similar prototype. All instances assigned to the same prototype compose a cluster. Each instance must belong to exactly one cluster. Once the clusters are assigned, a new prototype is calculated using the method previously discussed.

---

Ideally, the entire process described in the previous two frames is repeated until all prototypes remain static; however, if the process reaches 100 cycles, the existing prototypes are considered final and the process stops.

---

How many clusters can an attribute belong to?

- a - Two
- b - None
- c - One
- d - Less or equal to k

Answer: ugh

[complete]

---

Here's example code showing how to train K-prototypes `babyTrain(NULL,`

```
titanic, "kproto", k = 3, nstart = 1)
```

The first parameter is either our target attribute or a formula. Since K-prototypes needs neither, this gets set to NULL. `titanic` is a dataset, "kproto" tells `babyTrain()` to train a K-prototypes model, `k` is how many clusters we want (defaults to 3), `lambda` is the value we are multiplying the matching distance by (defaults to 1), and `nstart` tells `babyTrain` how many times we want to rerun the algorithm (defaults to 1)---more on that later.

---

Train k-prototypes on the iris dataset with `k = 4`. You don't need to assign the result to a variable or include any other parameters

```
babyTrain(NULL, iris, "kproto", 4)
```

*Below is the k-prototypes output.*

```
Numeric predictors:  4
```

```
Categorical predictors:  1
```

```
Lambda:  1
```

```
Number of Clusters:  4
```

```
Cluster sizes:  28 50 40 32
```

```
Within cluster error:  14.88958 24.08526 22.23344 22.39944
```

```
Cluster prototypes:
```

```
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

```
1 5.532143 2.635714 3.960714 1.228571 versicolor
```

```
2 5.006000 3.428000 1.462000 0.246000 setosa
```

```
3 6.252500 2.855000 4.815000 1.625000 versicolor
```

```
4 6.912500 3.100000 5.846875 2.131250 virginica
```

```
[complete]
```

---

Within cluster error is a measure of how well the prototype represents the cluster. When `k` remains constant, a lower within cluster error is preferred.

---

One way we can attempt to reduce this error is by running the algorithm

multiple times. Since the final prototypes can vary when we choose different initial prototypes, we can run the algorithm multiple times and choose the set of final prototypes with the lowest summed error. Setting the `nstart` parameter  $> 1$  will cause the algorithm to run multiple times, `babyTrain()` will then automatically return the set of prototypes with the lowest error.

---

```
Train k-prototypes on the iris dataset with k = 4. This time, use nstart = 10
```

```
skip
```

*This output omitted in appendix.*

```
[complete]
```

---

It looks like we get the exact same output as when running it once. Although improving our original prototypes would have been great, we now have some evidence suggesting that we are finding the best 3 prototypes we can. As an experiment, let's try turning `lambda` all the way down. This will essentially ignore categorical variables.

---

```
myModel <- . . . Train k-prototypes ONCE on the same dataset with k = 3 and lambda = 0. Assign the result to a new object called myModel.
```

```
skip
```

*This output omitted in appendix.*

```
[complete]
```

---

Aside from using the prototypes to gain insight into our dataset, we can also attach the cluster labels to the original dataset. To do this, we use `babyPredict()`. Just like when using it with a `knn` model, your model object is the first argument and a dataset is the second argument. However, instead of using a special unseen test set, we use the training set. This isn't really prediction (it's similar); we're just using `babyPredict()` to keep your workflow similar across algorithms.

---

```
use babyPredict() to attach cluster labels to the iris dataset.
```

```
babyPredict(myModel, iris)
```

*This output omitted in appendix.*

[complete]

---

It might be a little odd to assign labels to labeled data but it can be useful, especially for demonstration. You can see the assigned cluster on the far right and compare it to the species. The assigned cluster does a decent job of matching up with the species, so I conclude that biologists are right. These really are three species of flower. I also conclude the k-prptotypes module!

## A.8 Apriori algorithm

Apriori

---

Apriori is an algorithm for finding frequent itemsets. For our purposes, a frequent itemset is a commonly occurring set of row values/items. A set of row values may be a complete row/instance, it might be two items that frequently occur together---like high blood pressure and diabetes, or it could be a single item. The frequency of an itemset is called support. It's the number of times the itemset occurs divided by the total rows in the dataset. One parameter we set is the minimum acceptable support. This is used to ensure the only designated itemsets are ones we would consider frequent.

---

The number of items/columns is important criteria in determining what qualifies as an itemset

T/F: F

[complete]

---

The first step in the algorithm is to calculate the support of every value/item the dataset contains. As you can imagine, this is most meaningful on categorical data. For practical use on numeric data, numeric

values should be placed into bins defined by range. Once we calculate the support of the single items, we create candidate itemsets of size two by calculating all two item combinations of the single items with minimum support. We then calculate the support of the new itemsets and 'drop' those who do not meet minimum support from consideration. Those meeting minimum support are deemed frequent itemsets.

---

Once all size three itemset combinations are computed from the frequent size two itemsets, we can finally use what makes this algorithm special, the apriori principle. Instead of endlessly calculating support for all possible combinations---which requires many time consuming passes through the dataset---we can instead use the fact that all size  $n-1$  subsets of a size  $n$  itemset must minimum support for the size  $n$  itemset itself to meet minimum support. It's sort of like reaching the conclusion that diamond encrusted cellphones are rare, not because you counted them up, but because diamonds (a component of these phones) satisfy your criteria for rarity. This means that once we reach candidate itemsets where  $n = 3$ , we look at all of its  $n-1$  subsets and ensure that we had already determined them to be frequent. If any of its  $n-1$  subsets weren't frequent, then the size  $n$  set can't be frequent and we don't even bother calculating support. The size  $n$  candidate itemsets that pass the apriori consideration then have their support calculated from the dataset. This process repeats until we no longer find itemsets meeting minimum support.

---

What is the apriori principle (roughly)?

- a - According to the philosopher Ludwig Wittgenstein, the a priori principle . . . .
- b - Calculate the support for an itemset before calculating the support for its subsets.
- c - For something to be common, each of its components must also be common
- d - Useless

Answer: d

Answer: c

[complete]

---

The frequent itemsets found by Apriori are cool and all, but they are

usually used in the calculation of association rules. In fact, the paper that introduced Apriori is titled "Fast algorithms for mining association Rules" (Agrawal Srikant, 1994), despite only dedicating four sentences to the "sub-problem" of calculating association rules. Association rules are basically if-then statements; if we see the antecedent itemset, then we see the consequent itemset. There are two main measurements for the strength of an association rule. One is called confidence, the other is called lift. Confidence is the frequency of the entire itemset (antecedent and consequent together) divided by the frequency of the antecedent. It measures how confident we are in the consequent being present given we have the antecedent. Lift is (support of antecedent) / (support of antecedent \* support of consequent). Lift attempts to temper the effects of a consequent that is frequent throughout the entire dataset. This can lead to many rules with high confidence just because the consequent itself is frequent in general.

-----  
 Here's example of running Apriori: `babyTrain(NULL, flowerStuff, "apriori", minSup = 0.1, minConf = 2, maxLen = 10`

Like the previous examples, the parameters following the method ("apriori") are specific to the method and are shown set equal to their default value---the value used if you act like they don't exist at all. `minSup` is the minimum support. This can be increased to consider less total itemsets. `minConf` is the threshold below which a rule will not be shown to you. Consider Raising this if you're being presented with too many rules, or lowering if too few rules. `maxLen` is used for keeping the rule itself a reasonable size for your application.

-----  
 Run the default apriori settings on the arthritis dataset. don't assign a new object.

skip

*Below is a segment of apriori's output.*

```
1 Treatment=Treated, Age=[51,60.3) => Improved=Marked 0.1309524 0.8461538
2.538462 11
2 Treatment=Placebo, Age=[23,51) => Improved=None 0.1547619 0.8666667
1.733333 13
3 Age=[51,60.3), Improved=None => Treatment=Placebo 0.1190476 0.8333333
1.627907 10
```



[complete]

---

A numeric attribute (age) was detected, so the values were changed to ranges. You'll see a warning informing you of this action once the module is completed. This is a small dataset with few values, but we still found some useful rules. It looks like if you're treated and 51-60 yrs old, your arthritis is likely markedly improve. I think the other rules are pretty interesting too. Let's lower our standards to find some more.

---

This time, decrease minConf to 0.3 to see some more rules meeting the default minimum support of 0.1.

```
babyTrain(NULL, arthritis, "apriori", minConf = .3)
```

*This output omitted in appendix.*

[complete]

---

That's all I've got. This is the end of the Apriori module.

---

## A.9 Decision tree algorithm

### Decision Tree

---

Like the previously covered K-nearest neighbors, the Decision Tree algorithm is used for classification and regression. This algorithm recursively splits the data into two subsets based on how attribute values fall into buckets (temp < 70, yes/no?). The splits are chosen based on purity. If we only had As and Bs as target attribute values and a split was found that would create one subset of As and one subset of Bs, that split would be chosen. The buckets and the pattern used to create them are recorded and used for prediction.

---

First, the tree algorithm will attempt to find the best split. If used for classification (categorical target variable), the split leading to the smallest (weighted) gini index is used. The gini index of a dataset is  $1 - (\text{the sum of squared target class probabilities})$ . If we had a bucket with 4 red balls and 5 green balls, the gini index would be  $1 - (0.44^2 + 0.55^2) = 1 - (0.2 + 0.3) = 0.5$ . A bucket with 8 red balls and 1 green ball would have a gini index of  $1 - (0.01 + 0.079) = 0.2$ . It makes sense that the second bucket has a lower gini index and is therefore the preferred split. It's actually useful in making a prediction. It's a bucket that allows you to say, "If it's in here, it's likely a red ball". When building a tree, many pairs of subsets---the result of a split---will not share the ratio of instances equally. For example, subsets containing 3 and 7 instances in comparison to 5 and 5. Because of this, the Decision Tree algorithm weights the gini index of each subset by the ratio of instances it contains. The gini index from both subsets are then added together to create the gini index for the split. The split with the lowest gini index is chosen.

---

In babyCaret, splits are chosen based off of information gain

T/F: F

[complete]

---

If used for regression (numeric target variable), the split leading to the highest between-groups sum of squares is used. In general, a sum of squares is the sum of each value minus the mean; it's just the numerator when calculating variance. First, before any split is attempted, the target variable's grand mean is calculated. This is just the normal arithmetic mean of the pre-split set. After this, each value in a subset is 'temporarily replaced' by the subset mean and used to calculate a sum of squares between the each value in the subset and the grand mean. The split that maximises this value is then chosen. It's kinda like trying to maximize the numerator of an ANOVA's F-statistic. If everything is all cloudy to you, your get out of jail free-ish card is to think about it as trying to find a split which minimizes the p-value of its ANOVA.

---

Something something, sum of squares.

T/F: T

[complete]

---

If we let this process run continue until it no longer can, we're likely to end up with a bunch leaf nodes (ending buckets) that contain very small values. This is not necessarily a good thing. Our model may end up representing lots of nuances in the training set, but then fail to generalize to the population. This is called overfitting and it's something decision trees are prone to if we don't take steps to reduce their complexity. At our basic level, we have two main ways of doing so: we can set a lower threshold for the size of bucket the algorithm is allowed to split, and we can set the maximum depth---you'll see this later as the number of 'rows' in a plot. Lets check this out.

---

Here's example code showing how to train a Decision Tree:

```
babyTrain(titanic$survived, titanic, "tree", maxDepth = 30, minSplit = 2
```

As usual, the first parameter is our target variable, the second is the dataset it's from, and the third tells babyTrain() to train a Decision Tree model. maxDepth sets the maximum allowed depth (defaults to 30), and minSplit sets the bucket size under which a split will not be attempted (defaults to 2). The default tree parameters lead to a fully grown tree. This can be useful when you're only concerned with analyzing the particularities of a single dataset, but is likely to be overfitted and lacking in prediction accuracy.

---

```
myModel <- . . . Train a decision tree on the iris dataset with  
iris$Species as the target attribute. Use the default arguments for  
maxDepth and minSplit. Assign the result to a new data object called  
myModel.
```

```
myModel <- babyTrain(iris$Species, iris, "tree")
```

*Plot shown in RStudio.*

```
[complete]
```

---

Check out the plot in the lower right window. It looks . . . complicated. It's likely overfitted to the training data. This will cause the model to capture lots of detail from the training set but fail to generalize to other datasets. Let's test the model accuracy.

---

Use `babyPredict()` to evaluate accuracy on `testSet`

skip

```
[1] "Percent misclassified: 7.69"
```

```
[complete]
```

---

We might be able to get a more accurate model if we reduce the complexity of the model. We'll do this by increasing `minSplit`.

---

```
myModel <- . . . Train the same model and re-assign to myModel, but this  
time set minSplit to 25
```

skip

```
[complete]
```

---

```
myModel <- . . . Train the same model and re-assign to myModel, but this  
time set maxDepth to 3
```

skip

*Plot shown in RStudio.*

```
[complete]
```

---

That's better. If you look at the first split chosen at the very top, you can see that we were able to include all of the `setosa` flowers into one subset.

That looks better. If you look at the first split chosen at the very top, you can see that we were able to include all of the `setosa` flowers into one subset. This was chosen due to the resulting low gini index. Next, we'll check the accuracy.

---

Use `babyPredict()` to evaluate accuracy on `testSet`

skip

```
[1] "Percent misclassified: 3.84%"
```

```
[complete]
```

---

Great! We've increased the model's accuracy. Since we've reduced the model's complexity, it's better able to generalize to unseen data. It's a better representation of how the patterns in the data can be used for species classification. Now let's try using decision tree for regression. Once we get a good regression model, we'll end this module.

---

```
myModel <- . . . Train a decision tree on trainSet with
trainSet$Petal.Length as the target attribute. Use the default arguments
for maxDepth and minSplit. Assign the result to myModel.
```

```
myModel <- babyTrain(trainSet$Petal.Length, trainSet, "tree")
```

*Output omitted from appendix due to length.*

```
[complete]
```

---

This tree is so complicated that plotting it is likely to crash your R session. Instead of plotting, I've printed a text based representation of the tree to your console. If you scroll up through the output, there should be no doubt in your mind that the model is overfitted. Let's check accuracy on the test set.

---

```
Use babyPredict() to evaluate accuracy on testSet
```

```
skip
```

```
[1] "Mean absolute percentage error: 33.3"
```

```
[complete]
```

---

We can definitely do better. We'll reduce model complexity by decreasing maxDepth (defaults to 30).

---

```
myModel <- . . . Train the same model and re-assign to myModel, but this
time set maxDepth to 2
```

```
myModel <- babyTrain(trainSet$Petal.Length, trainSet, "tree", maxDepth = 2)
```

```
[complete]
```

---

Use `babyPredict()` to evaluate accuracy on `testSet`

```
babyPredict(myModel, testSet, TRUE)
```

```
[1] "Mean absolute percentage error: 11.5%"
```

```
[complete]
```

---

Awesome! I think that's accurate enough to end the decision tree module.

## Appendix B

### Materials for Future Formal User Feedback

This section of the appendix contains materials that may be used for future testing and evaluation of the tutorials by users.

#### B.1 Help Sheet

##### Entering Commands

- Type a line of code in the console then press enter to execute that command.

##### Some Commands

```
tutorial()
```

This will begin the tutorial. This should be the first line of code that you run.

**Note:** If you use any of the following commands, please let the researcher know that you used and why you used it.

```
skip
```

This will skip the current frame of the tutorial. If you are stuck on an action due either to a bug or not knowing the answer, this command should move you past it. Please let the

researcher know whenever you use skip and why you used skip.

```
exit
```

This command will kick you out of the tutorial and return you to the normal R console. This will end your tutorial session. NOTE: please do not X out of RStudio.

## Some Tips

- If you can't remember how to do something you've previously done, you can scroll up through the console to see the questions asked and how you responded. Use the scroll wheel for this rather than the arrow keys.
- You should primarily be using the *console* window of R studio; it's on the left side of the screen. You will not need the source or job tabs. You will occasionally interact with the two windows on the right side of the screen. To the extent you do, you will only be using the lower section and the *environment* tab of the upper section.

## Your Goal

Your goal is to complete the tutorial while understanding the information presented. Run the `tutorial()` command in the console window when you are ready.



## B.2 User Survey

1. What is your major?

2. What year are you in school?

3. What is your experience in using the programming language R?

- None
                         
  Comfortable
                         
  Expert  
 Familiar
                         
  Competent  
 Some experience
                         
  Advanced

If you **did not** answer *none*, would you say that you primarily use R for statistics/data-analysis?       Yes  No

4. What programming language(s) do you know and what is your experience level?

Language	None	Familiar	Some Experience	Comfortable	Competent	Advanced	Expert
C/C++							
C#							
Java							
Python							
Other							

5. What is your greatest experience with machine learning software that you are most comfortable with?

- None                       Comfortable                       Expert  
 Familiar                       Competent  
 Some experience                       Advanced

If you **did not** answer *none*, what is the software? If it's a programming language/-package, please include the language *and* package.

.....  
 .....

6. Have you ever completed a programming oriented course in any of the following disciplines: computer science, information technology, electrical engineering, mathematics, or computer information science?                       Yes  No

7. How familiar are you with the following machine learning algorithms?

(a) K-nearest neighbors

- I didn't know it existed.                       I understand it on both applied and technical levels.  
 I only know it exists.                       I would consider my understanding deep.  
 I understand it on an applied level.

Have you computed or instructed software to compute this algorithm before?

- Yes  No

(b) K-prototypes (K-means with a hamming distance for categorical variables)

- |   |  |
|---|--|
| <input type="checkbox"/> I didn't know it existed.            | <input type="checkbox"/> I understand it on both applied and technical levels. |
| <input type="checkbox"/> I only know it exists.               | <input type="checkbox"/> I would consider my understanding deep.               |
| <input type="checkbox"/> I understand it on an applied level. |  |

Have you computed or instructed software to compute this algorithm before?

- Yes  No

(c) Decision tree

- |   |  |
|---|--|
| <input type="checkbox"/> I didn't know it existed.            | <input type="checkbox"/> I understand it on both applied and technical levels. |
| <input type="checkbox"/> I only know it exists.               | <input type="checkbox"/> I would consider my understanding deep.               |
| <input type="checkbox"/> I understand it on an applied level. |  |

Have you computed or instructed software to compute this algorithm before?

- Yes  No

(d) Apriori

- |   |  |
|---|--|
| <input type="checkbox"/> I didn't know it existed.            | <input type="checkbox"/> I understand it on both applied and technical levels. |
| <input type="checkbox"/> I only know it exists.               | <input type="checkbox"/> I would consider my understanding deep.               |
| <input type="checkbox"/> I understand it on an applied level. |  |

Have you computed or instructed software to compute this algorithm before?

- Yes  No

Stop! Please do not proceed until you have ceased working on the tutorial.

- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
1. I would use this tutorial again
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
2. I would use future versions of this tutorial
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
3. I would use the R package again
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
4. At times, I felt frustrated with the tutorial
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
5. I feel that the tutorials aided my learning
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
6. I had difficulty navigating the tutorial
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
7. I feel that bugs or issues with the tutorial hindered my learning
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
8. I feel that bugs or issues with the R software hindered my learning
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
9. I would recommend this to someone unfamiliar with programming
- |   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
10. I would recommend this to someone familiar with programming
11. What about the tutorial helped you learn to use the software?

.....

.....

.....

.....

12. What would help your learning in this software environment?

.....  
.....  
.....  
.....

13. What did you like about the tutorial?

.....  
.....  
.....  
.....

14. What did you dislike about the tutorial?

.....  
.....  
.....  
.....

15. Do you have any suggestions for improving your experience with the tutorial?

.....  
.....  
.....  
.....



Thank you for participating in this research. If you want more information about the tutorials, please ask!

## B.3 Researcher Instructions

### 1. Consent form

- (a) Explain the study.
- (b) Give participant the consent form to read and sign.
- (c) Ensure they have initialed every page and signed the last one.
- (d) Ask if they have any questions.

### 2. Survey

- (a) Give them a survey and instruct them to fill out first section.
- (b) Open RStudio software.
- (c) Load package.
- (d) Maximize console window.
- (e) Clear the console.

### 3. Help Sheet

- (a) Read through and explain the help sheet.
- (b) Ask if they have any questions.



- (c) Ask them to let you know if they get stuck or have difficulties.
- (d) Remind them know you will be taking notes (as written in the consent form).
- (e) Instruct them to run the tutorial() command and begin the tutorial.
- (f) Note their start time.

#### 4. During Task

- (a) If participant gave email, send them a copy of the consent form.
- (b) Stop them 30 minutes after their start time or if they complete the tutorial.
  - If they are having difficulties related to the content you may suggest they scroll up through the console and reread, or that they may use the skip command.
  - If they are having difficulties not related to the content e.g. they minimize the console window. You may resolve the issue for them.
  - Take notes on what feedback the participant gets stuck on or has questions on. If they use “skip”, or “exit” ask and record why they chose to use that command.

#### 5. After Task

- (a) Have them complete the survey
- (b) Ask if they have any questions.
- (c) let them know they can complete the tutorial if they would like during open lab hours.
- (d) Thank them for their participation.

## Appendix C

### Public Functions

This section contains short descriptions of **babyCaret**'s public functions. Where a parameter is set equal to a value, the value signifies the default argument when used when the user does not specify a value.

#### 1. `babyTrain(formOrY, data, method, ...)`

`babyTrain()` handles ML model training through interaction with **babyCaret**'s ML model implementations.

- `formOrY` accepts either the dependent variable as a vector, or R's implementation of Wilkinson-Rogers notation [32]. These are two ways for the user to specify a target attribute. `formOrY` accepts NULL for apriori and  $k$ -prototypes methods.
- `data` accepts a dataframe containing the training data.
- `method` is a string specifying the ML algorithm to be trained. `knn`, `kproto`, `tree`, `apriori`, or `hmm`. `Hmm` is a fake model used for tutorial purposes.
- `...` contains model specific parameters.
  - `knn` has  $k = 3$ .
  - `kproto` has  $k = 3$ , `max.iter` = 100, and `nstart` = 1.
  - `tree` has `maxDepth` = 30 and `minSplit` = 2.

- apriori has  $\text{minSup} = 0.1$ ,  $\text{minConf} = 0.8$ , and  $\text{maxLen} = 100$ .

## 2. `babyPredict(trainedModel, newdata, istest = FALSE)`

`babyPredict()` predicts missing values through via a model trained by `babyTrain()`.

- `trainedModel` is a model returned by `babyTrain`.
- `newdata` is the dataframe containing values to be predicted.
- `istest` is a Boolean value specifying whether `newdata` has known target values and is being used to assess model accuracy. If `TRUE`, prints information on model accuracy, if `FALSE`, predicts `newdata`'s missing target values

## 3. `dataPartition(df, p = 0.75)`

`dataPartition()` splits a single dataset into list containing a training set and a testing set.

- `df` is the dataframe being split.
- `p` is a parameter between 0 and 1 governing the ratio of data being sent to the training set.  $(1-p) \cdot 100\%$  of data is sent to the testing set.

## 4. `(plotTree(tree, showInterior = FALSE))`

`plotTree()` plots a decision tree using R's `rpart.plot` package [19]

- `tree` is a decision tree model returned by `babyTrain()`.
- `showInterior` is a Boolean value that when `TRUE`, will include non-terminal nodes in the plot.

## 5. `tutorial()`

`tutorial()` opens the tutorial menu inside the user's R console.