$$N_A\left(X^{t+1}\right)=\left\{X=\mu\circ X^t \ s.t.\,\text{LastUsed}\left(\mu^{-1}\right)<\left(t-T\right)\right\}, X^{t+1}$$
$$=\text{Best}-\text{Neighbor}\left(N_A\left(X^t\right)\right)\}$$

For basic moves acting on binary strings, $\mu=\mu^{-1}$.

For stochastic models, we can substitute prohibition rules with probabilistic generation-acceptance rules with large probability for allowed moves, and small for prohibited ones. Stochasticity can increase the robust nature of TS algorithms. Stochasticity can limit or remove the benefit of memory-induced activity, as is the main draw to tabu search. *Robust tabu search* features a prohibition parameter that is randomly changed between an upper and lower bound during the search. In *fixed tabu search*, stochasticity can be added by randomly breaking ties, or the cost function decrease is obtained by more than one candidate of the Best-Neighbor() function. This same effect is observed when implementing stochasticity in reactive tabu search.

Fixed Tabu Search

Let us assume we have a search space *X* such that $X=\left[b_1, b_2 b_3\right]$ with a cost function $f\left(\left[b_1,b_2,b_3\right]\right)=b_1+2b_2+3b_3=7b_1b_2b_3$, where *b* is a 3-bit string. The feasible points will be the edges of the 3-dimensional cube shown in Figure 8-8. The neighborhood of a given point is the set of points that are connected with edges. The point X^0 = [0,0,0] with f(X^0) = 0 is a local minimizer due to the fact that other moves produce a higher cost.

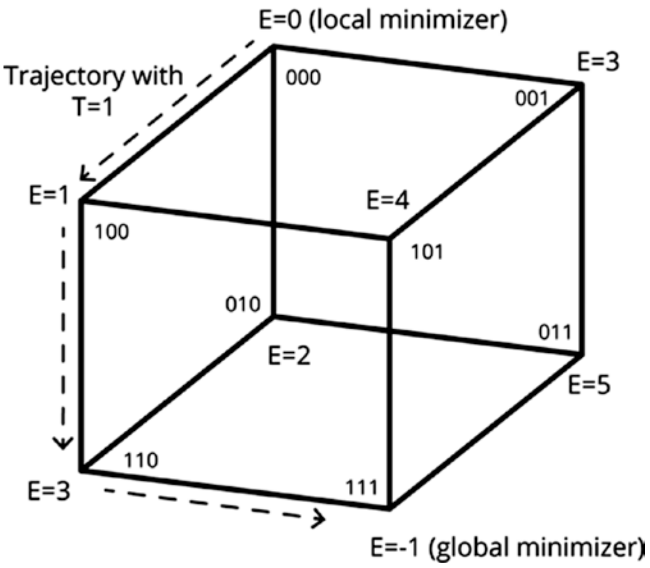


Figure 8-8. *A feature space with error function, E, and f value = [x,y,z], using tabu search*

We will define two parameters that will be of use to testing the efficiency of a given tabu search epoch, denoted as the Hamming distance and the minimum repetition interval. The *Hamming distance* describes the distance between the starting point and the most successful point along the search trajectory, and the *minimum repetition interval* describes the amount of times a similar move was visited along a given search trajectory. These parameters' equations are given by the following:

$$H\left(X^{t+1}, X^{t}\right) = \tau, \quad \tau \leq T + 1,$$

$$X^{t+R} = X^{t} \Rightarrow R \geq 2(T+1)$$

Moving forward, we should direct our attention to avoiding attractors of the search trajectory, where we define *attractors* as local minima generated by deterministic local search. If the cost function is lower bounded, and starts from an arbitrary point, it will terminate at local minimizer. We also define what is known as an attraction basin. An *attraction basin* is composed of all points such that a deterministic local search trajectory starting from them terminates at a specific local minimizer. Deterministic search trajectories often suffer from being biased towards attraction basins and as such can yield a result that is not a global minimizer. To solve this, a given search point is kept close to a local minimizer that was found in the beginning of the search trajectory. After this, the search trajectory can search for better attraction basins with respect to reducing the cost function. As always, there are limitations that we must be conscious of. With tabu search, the difficulties that are most frequently encountered are the determination of an appropriate prohibition parameter and making the technique robust enough that it doesn't require tedious amounts of tuning from one context to another. This brings us to reactive tabu search, which has been proposed as a method of solving these problems.

Reactive Tabu Search (RTS)

Reactive tabu search (RTS) features a prohibition parameter that is determined through reactive mechanisms within the search trajectory. We initialize it with a value of 1 in the very beginning, but we add a deterministic aspect to how it changes. If there is evidence that diversification in the search trajectory is needed, T increases. Once this evidence isn't apparent, T decreases. Sufficient evidence for diversification in the search path is reached when we repetitively visit previous points along the search trajectory, as they are stored in the "memory" of the algorithm. Also, to avoid instances in which the algorithm is very rigidly stuck in an attraction basin, RTS has an escape mechanism. This is initiated when too many search trajectory configurations have been repeated in a given period and features a stochastic reconfiguration of the current search path.

The objective function, *f,* ultimately is where the information for the direction of the search trajectory comes from. As such, the following algorithms directly fall under this paradigm.

WalkSAT Algorithm

The WalkSAT algorithm can be understood as a more generalized version of the GSAT algorithm, which is a type of local search algorithm. In the algorithm, there are a set number of opportunities allowed for a given number of iterations to find a solution. During a given iteration, the algorithm chooses a variable between two criteria. After this point, the variable is put into the FLIP function where $\mathrm{FLIP}(\mathrm{x}_\mathrm{i}) = (1 - x_i)$. The WalkSAT gets its power from doing less calculation than GSAT because it is considering fewer parameters at a given time. In addition to this, by a product of the clauses which determine variable picking, it thereby has the opportunity to solve a problem variable that could be preventing convergence upon the global optimum. Clause-weighting can also be incorporated into the WalkSAT algorithm, which gives new possibilities for parameter tuning and feedback loops produced upon, The following algorithm suggests weights as a method of encouraging more priority on solving the more difficult clauses. Difficult clauses are considered such after several configurations.

K-Nearest Neighbors (KNN)

KNN is considered to be instance-based learning, which features approximations of the function locally and all calculations happening after classification. It can also be used for regression, but often is described as a search method. Its main draws are the fact that it is relatively easy-to-understand and effective for cases in which there are irregularities in the pattern of data. These models, in the case of classification, are considered memory-based where we define k neighboring points that we want to consider. We use a Euclidean norm on the standardized data to determine the distance between a given point and its k neighbors. This equation is given as

$$d(x,y) = \sum_{i=1}^{N} \sqrt{(x_i - y)^2}$$

where $I = 1,2,\ldots,N$, N = the total number of observations, x_i = ith observation, and y = the specific point we want to classify.

As K increases, typically we notice that the definition between classes becomes less rigid, leading to generally more robust models. Insofar as it relates to feature selection, KNN can be used as a data preprocessing technique often used alongside other search techniques for more refined feature selection. An example is given from a 2007 paper by Tahir, Bouridane, and Kurugollu in which they create a hybrid algorithm using a variant of tabu search and KNNs. The algorithm performs feature weighting and selection, yielding more accurate classification results. The pipeline occurs such that the features are selected and weighted via tabu search and classified via KNN. If we don't perform feature selection with tabu search, or feature selection at all, more noise is incorporated into the decision-making process for the KNN algorithm. As the case generally is, performing feature selection here helps the algorithm make more precise choices when classifying each observation.

Summary

This chapter was a kind of meta-heuristic on the entirety of the granular details discussed up until this point. Foremost, experimental design, feature selection, and A/B testing will be crucial to any data scientist's profession. The ability to properly structure the experiments by which you conduct models, improve upon their performance by modifying the inputs, and then quantitatively validate the results of a model are crucial. Chapter 9 discussed hardware solutions for those who are interested in creating a build for personal or professional use.

CHAPTER 9

Hardware and Software Suggestions

To apply the techniques explored in this book in a professional setting, hardware upgrades may become a consideration. In some cases, it might even be necessary to build a computer from the ground up. There are very few out-of-the-box ready builds, and the ones that do exist can cost a staggering amount of money. With that in mind, this chapter is intended to give readers a basic overview of the hardware components they should be most mindful of as well as provide general suggestions on hardware to purchase.

Processing Data with Standard Hardware

You may face many difficulties when operating on a relatively "vanilla" machine. When working on machine learning and deep learning problems with a large data set, it is generally recommended that you run most of your operations on subsets of the data and train in such a manner that the iterations times the size of the subset equals the size of the original data set. Although this merely provides an approximation of performance, it may be able to run your solution without crashing the interpreter due to lack of RAM.

It is also highly suggested that individuals with sufficient funds use Amazon Web Services (AWS). Professionally, Amazon is the go-to solution for cloud services and may even allow you to pick up a valuable skill set that many employers are eager to have. In short, you can pay to run instances of all the hardware you need in a cloud environment. Although for deployment purposes doing so can be extremely costly, for proof of concept or research using a cloud service like Amazon AWS can be a cost-efficient and easy solution to solving your problems for deep learning. If you need to implementing solutions as part of deploying an algorithm for a business or service, however, read on—the advice given in this chapter is a good starting point.

Solid State Drives and Hard Drive Disks (HDD)

A *hard drive disk* (HDD) is a storage device used to retain information even while the machine is not online. The main characteristics of an HDD are the amount of data it can store and the performance it provides. Since the mid 2000s, as I mentioned early

© Taweh Beysolow II 2017
T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_9

in the book, the price of storage has dropped substantially, promoting a resurgence in interest in the science of machine learning and deep learning. This development makes it possible to store and collect substantial amounts of training data and/or trained models that you can update later moments or use for related tasks. Users should become familiar with the cases they want to tackle most often.

Graphics Processing Unit (GPU)

GPUs are one of the most frequently referenced pieces of hardware with respect to distinguishing machines that can deliver high-performance deep learning from machines that aren't specialized for deep learning). For deep learning, GPUs accelerate the processing of computations and are an integral part of the deep learning build. When compared with Central Processing Unit (CPU) computations, GPUs easily outperform CPUs and are where the bulk of computation occurs. You can build a unit with multiple GPUs, but be aware of the challenge of efficiently utilizing computing power when you do this. If you're not familiar with parallel computing, achieving such a build can be time-consuming to learn and implement correctly and invites spending an unknown amount of time, not to mention the time involved in designing and debugging software before algorithms/solutions can be effectively deployed.

There are packages in different languages to parallelize your code and improve performance. In R, I suggest you consider the parallel package, especially for performing the same task on a large amount of data. Rather than inputting the whole data set into an algorithm, it can be broken up such that the same task is performed in parallel with chunks of the data set, thereby making it more efficient. Where applicable, you should also implement the `lapply` function. This function takes a parameter and feeds it into a function, making performing complex operations much more computationally efficient than using nested loops.

My recommendations for GPUs (as of early 2017) focus on the following Nvidia models:

- Titan X
- GTX 680
- GTX 980

As of early 2017, Nvidia is one of the few companies devoting attention to developing GPUs specifically for the purpose of deep learning. (Note that AMD is partnering with Google to create deep learning hardware to be released sometime in 2017.) While this is likely not to be cost-effective for the average practitioner, for those in a professional context or with sufficient budgets, I suggest you review the specifications and performance reviews for AMD's FirePro S9300 x2 GPU when it releases.

Choosing a GPU depends on the type of problem you want to solve and how much memory you expect to consume in the process. Those using CNNs should expect to consume a great deal of memory, particularly in the process of training a given model. The physical storage for images and other data with deep learning is another consideration to keep in mind. Though both solid storage and virtual storage have dropped in price dramatically, you should set aside time to properly estimate the storage necessary.

Central Processing Unit (CPU)

The CPU instructs the computer on what operations should happen and where these operations should happen, in addition to performing very basic arithmetic, logical, and input/output functions. The CPU also works closely with the GPU to initiate function calls and initiate transfers of computations to the GPU. For deep learning–specific work, the number of CPU cores as well as CPU cache size are important. Most deep learning libraries rely on using a single CPU thread, and you can often perform just fine with one thread per GPU. However, using more threads per GPU will likely lead to better performance—take this fact in context with the task you intend to perform. For image-classification tasks, such the classic MNIST digit-classification task, I have found that using g2.2xlarge instances from AWS is more than sufficient, if I have difficulty using my local machine—it provides 1 GPU with 15 GB of RAM and 60 GB of SSD storage.

With respect to CPU cache size, there are several cache types with varying speeds. L1 and L2 tend to be quick, and L3 and L4 are slow. The purpose of the CPU cache is to help speed up computation via matching a key pair value. Most data sets encountered in a practical context are too large to fit into a CPU cache, so new data will be read in from the RAM on a given computer for each mini-batch. In the case of deep learning, most of the computation takes place in the GPU, so you needn't worry about buying CPUs that can handle this load. However, due to CPU cache misses, you may often see that the machine underperforms and you have latency issues. That leads to the core consideration with cache misses: RAM and the need for more of it so often in machine learning and deep learning.

Random Access Memory (RAM)

RAM stores frequently used program instructions such that the speed of programs increases because it stores data that will be read or written irrespective of its location within the RAM. As for the size of RAM you need, it should be comparable to the size of the GPU you're using. Using less RAM than the size of the GPU is likely to lead to latency issues that can cause problems particularly when training different networks such as CNNs. Using more RAM rather than less allows you to perform preprocessing and feature engineering much more easily than otherwise. It's easy to say, "Buy as much RAM as possible," but of course that's not always possible. However, you should consider investing a significant portion of available capital in this aspect.

Motherboard

The motherboard is the main circuit board, found in a variety of products besides personal computers. Its primary purpose is to facilitate communication between various components within a computer, and it holds the connectors between these components. Make sure the motherboard has enough PCIe ports to support the number of GPUs that will be installed in a given computer, as well as support all the other hardware components being chosen.

Power Supply Unit (PSU)

Power supply units convert alternating current electricity to regulated direct current power so that it can be used by the components within the computer. With regard to PSUs used for deep learning, be mindful to buy one that can service the number of GPUs you use if you use more than one. Deep learning can often require intensive periods of training, and the costs of running these instances should be minimized. The required watts for a given deep learning machine can be approximated by summing the watts of the GPU and CPUs while adding roughly 200 watts for the other components within the computer and variances in power consumption.

Optimizing Machine Learning Software

The major purpose of this chapter is to allow the reader to find where to focus their attention with respect to improving their machine. The end goal is to improve the performance of the software being tested and deployed, but part of that involves optimizing the software directly. To that end, before all other steps, I advise you to try to improve the algorithm you're using or find a better one when implementing a given solution. Optimal choice of algorithm and finding the most optimal implementation of said algorithm can be quite time-consuming. It might involve reading through a considerable amount of documentation, looking through the code for various functions in depth, and possibly doing experimentation. Although this book is intended for those who are relatively experienced in R and who are new to deep learning/machine learning, after reading through this text you should feel confident enough to begin creating your own implementations of various machine learning algorithms. Although time-intensive, doing so can teach you a great deal about the efficiency of different algorithms and their implementations.

A common debate currently revolves around which language to use. R is a very accessible language and great for proof of concept, particularly because its syntax allows for code to be written and tested quickly. Yet it can often prove cumbersome when trying to deploy the algorithms for anything that requires real-time applications—and particularly when trying to embed the software into other applications. If you intend on working in a professional context, keep that in mind when devising final solutions for anything. Typically, those looking to write for speed often do so in C++. This book doesn't cover C++, of course, or any of the packages in C++ for that matter, but readers should explore the myriad of libraries available in C++ for machine learning and deep learning.

Summary

This chapter should give readers a basic understanding of some of the most common concerns they should have when making a dedicated build for machine learning—or when trying to modify their existing hardware to better service their deep learning needs.

Chapter 10 dives into practical examples more heavily using machine learning and deep learning solutions.

CHAPTER 10

Machine Learning Example Problems

In this chapter we'll start applying the techniques discussed so far to practical problems you may potentially face. The data sets provided will either be generated from random data or will be from `https://github.com/TawehBeysolowII/AnIntroductionToDeepLearning`. Note that you can also consult that URL for all code and data sets provided in the examples given in prior chapters.

In this chapter we will be exclusively examining machine learning problems. Though I can't cover every possible field and problem type, the focus on the examples here will to be address common scenarios users are likely to encounter.

I encourage you to view these final example chapters as tutorials for how to go from a data set (raw or processed) to a solution. Although these examples are feasible solutions, the most important aspect is applying the experimental design, feature selection, and model evaluation methodologies we've already discussed to solve problems effectively.

Problem 1: Asset Price Prediction

Quantitative finance is a field that continues to incorporate data science and machine learning techniques into its methods, specifically in the process of automated trading and market research. Although quantitative finance in and of itself is a field with a rich diversity and its own techniques, there are many broad analytic and mathematical concepts we can apply. For this example, we will be using the quantmod package to download financial data, and I'll walk you through how to predict asset prices. I'll also briefly explain how to create a trading strategy—specifically, a statistical arbitrage strategy. As always, backtesting these results is highly recommended prior to anyone applying these techniques. The purpose of this chapter is to provide an academic understanding of machine learning—it's not intended as a tutorial in quantitative portfolio management!

Let's assume you're a quantitative analyst at an asset management firm and you're tasked with reasonably predicting the returns of an asset that is in the S&P 500. Your managing director believes that there are ten other stocks that would be helpful in

© Taweh Beysolow II 2017

modeling the performance of this particular asset and that you should likely somehow use these in your analysis. The director gives no prescriptions particularly on what to use, besides suggesting using a machine learning approach to solving this problem.

Let's begin by defining the problem.

Problem Type: Supervised Learning—Regression

Any problem in which we're trying to predict discrete or continuous values is known as a *regression* problem. Because we have the answers, and we're trying to compare our proposed answers against the actual answers, this is a supervised learning problem. Specifically, we'll be trying to predict the returns of one asset, y, based on the returns of other assets, x. Let's start building the pipeline to solve this problem.

Typically, using the Yahoo! or Google Finance API is recommended for these tasks. For those particularly focused on the application of machine learning to quantitative finance, note that Yahoo! Finance's data has survivorship bias built in—that is, any companies that are now defunct cannot have their data accessed anymore. So, companies that were delisted for any reason are no longer stored in the database. This creates a problem because all strategies using this data won't reflect the worst possible downside had someone, for example, traded securities such as Bear Sterns of Lehman Brothers during the financial crisis. However, databases that hold data of companies that went bankrupt or are no longer listed can be found (Norgate Data is one example).

We'll begin by loading data using the Google Finance API, but will do so using the quantmod package. This package is recommended for any work requiring access to stock data, such as getting daily, monthly, or quarterly prices for various financial instruments, in addition to getting data on financial statements from publically listed companies.

Let's start walking through the code:

```
#Clear the workspace (1)
rm(list = ls())

#Upload the necessary packages (2)
require(quantmod)
require(MASS)
require(LiblineaR)
require(rpart)
require(mlbench)
require(caret)
require(lmridge)
require(e1071)
require(Metrics)
require(h2o)
require(class)
#Please access github to see the rest of the required packages!
```

```
#Summary Statistics Function
#We will use this later to evaluate our model performance (3)
summaryStatistics <- function(array){
  Mean <- mean(array)
  Std <- sd(array)
  Min <- min(array)
  Max <- max(array)
  Range <- Max - Min
  output <- data.frame("Mean" = Mean, "Std Dev" = Std, "Min" =  Min,
  "Max" = Max, "Range" = Range)
  return(output)
}
```

In the preceding code, as always when using R, it's important to clear the workspace (1) when working with a new experiment. Then we load the required packages (2). The next function defined gives summary statistics on the arrays that we're analyzing (3). In this example, we'll be looking exclusively at MSE. This is to provide a simple example of how to evaluate machine learning models.

There are two approaches I often take:

- Evaluate several models in default mode and then perform parameter tuning on the best model.
- Perform parameter tuning one parameter at a time and then evaluate the tuned models against one another.

Here, I'll be performing the latter, though to a less intensive degree for the purpose of simplicity and explanation.

Description of the Experiment

The general pipeline we will create to solve this problem can be described as follows:

Data Ingestion → Feature Selection → Model Training and Evaluation → Model Selection

Specifically, in this problem we will try to predict the returns of Ford, ticker F, based on the returns of stocks we suspect accurately describe these returns (a mix of market indices and other stocks). The selection of our stock portfolio could be a study in and of itself, but in this instance we chose stocks that are related to the auto market (macro indicators and those tied to the energy industry). The assumption here is that stocks that track the performance of Ford are likely to be companies within the same industry, in related industries that service the auto market in some way, or describe greater implications about the economy at large.

Be aware that beyond the mathematics necessary to properly understand how to create machine learning models, it's necessary to provide these models with useful data. If we were to use features that are completely irrelevant to the problem being solved, we would be very unlikely to receive any useful results as output from a fitted model. As such, these assumptions we made to create our data set will help yield the better results prior to any fine tuning we perform on our algorithms:

```
#Loading Data From Yahoo Finance (4)
stocks <- c("F", "SPY", "DJIA", "HAL", "MSFT", "SWN", "SJM", "SLG", "STJ")
stockData  <- list()
for(i in stocks){
  stockData[[i]] <- getSymbols(i, src = 'google', auto.assign = FALSE, from
= "2013-01-01", to = "2017-01-01")
}

#Creating Matrix of close prices
df  <- matrix(nrow = nrow(stockData[[1]]), ncol = length(stockData))
for (i in 1:length(stockData)){
  df[,i]  <- stockData[[i]][,4]
}
#Calculating Returns
return_df  <- matrix(nrow = nrow(df), ncol = ncol(df))
for (j in 1:ncol(return_df)){
  for(i in 1:nrow(return_df) - 1){
    return_df[i,j]  <- (df[i+1, j]/df[i,j]) - 1
  }
}
```

In the preceding code, we pull the data from Yahoo! Finance (4). Unless this data is saved after initial download, you should have an active Internet connection—otherwise this part of the code won't execute properly. When calculating the returns of a given stock, you can think of returns as a derivative, but a simpler formula for a return based price is the following:

$$Adjusted\ \ CloseR_{x_t} = \left(\frac{P_{x_t+1}}{P_{x_t}}\right) - 1$$

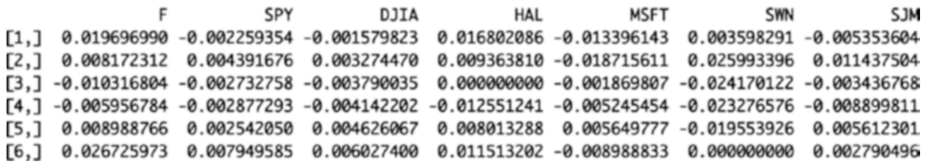
(A)

$$Where\ x = stock\ x,\ y = stock\ y,\ t = time\ period\ (1,2,\ldots\ n),$$

$$n = number\ of\ observations,\ and\ \ P_{x_t}\ = Price\ of\ Stock\ x\ in\ period\ t$$

For the purpose of this experiment, and likewise in many such cases in quantitative finance, we calculate returns based on adjusted close prices (equation A). We call these *adjusted close prices* based on their reflecting any changes in the underlying stock price over time due to dividends, stock splits, or other financial adjustments that have nothing to do with the performance of the stock or market conditions. Here, we will be looking at daily returns. The selection of the time frequency is entirely up to the user and depends on the strategy being assessed. Generally speaking, high-frequency trading occurs multiple times within a day, and low-frequency trading occurs in increments significantly longer than a day.

We organize the data such that each column represents the returns of a given stock and each row represents the return on a given day. Figure 10-1 shows the head of the data set.



```
                F          SPY         DJIA          HAL         MSFT          SWN          SJM
[1,]  0.019696990 -0.002259354 -0.001579823  0.016802086 -0.013396143  0.003598291 -0.005353604
[2,]  0.008172312  0.004391676  0.003274470  0.009363810 -0.018715611  0.025993396  0.011437504
[3,] -0.010316804 -0.002732758 -0.003790035  0.000000000 -0.001869807 -0.024170122 -0.003436768
[4,] -0.005956784 -0.002877293 -0.004142202 -0.012551241 -0.005245454 -0.023276576 -0.008899811
[5,]  0.008988766  0.002542050  0.004626067  0.008013288  0.005649777 -0.019553926  0.005612301
[6,]  0.026725973  0.007949585  0.006027400  0.011513202 -0.008988833  0.000000000  0.002790496
```

Figure 10-1. *Head of stock return data set*

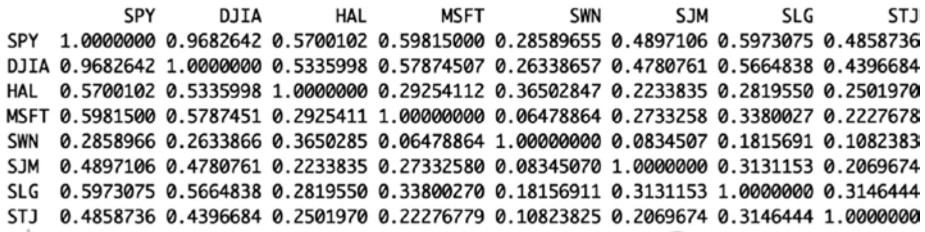
Stock returns often work well with machine learning algorithms because they are all scaled similarly and represent a measure that is relative to all the observations within a given stock, as well as the universe of stocks available for analysis.

Feature Selection

When handling time series data, we often encounter multicollinearity. Because of this, PCA is a fair method to use for feature selection. We do so because there are likely features that are unnecessary to evaluate, and therefore noise need not be valuated, in addition to the fact that linear correlations among variables are high. So, evaluating features by their variance contributed is reasonable. The following shows the code that performs PCA:

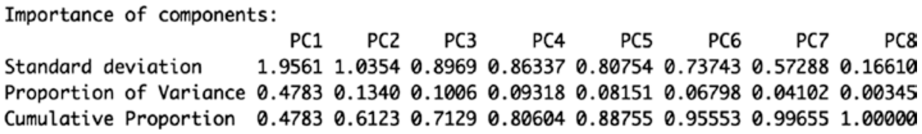
```
#Feature Selection
#Removing last row since it is an NA VALUE
return_df  <- return_df[-nrow(return_df), ]
#Making DataFrame with all values except label IE all columns except for
Ford since we are trying to predict this
#Determing Which Variables Are Unnecessary
pca_df  <- return_df[, -1]
pca  <- prcomp(scale(pca_df))
cor(return_df[, -1])
summary(pca)
```

When executing the preceding code, we receive the results shown in Figures 10-2 and 10-3.



```
           SPY      DJIA       HAL       MSFT        SWN       SJM       SLG       STJ
SPY  1.0000000 0.9682642 0.5700102 0.59815000 0.28589655 0.4897106 0.5973075 0.4858736
DJIA 0.9682642 1.0000000 0.5335998 0.57874507 0.26338657 0.4780761 0.5664838 0.4396684
HAL  0.5700102 0.5335998 1.0000000 0.29254112 0.36502847 0.2233835 0.2819550 0.2501970
MSFT 0.5981500 0.5787451 0.2925411 1.00000000 0.06478864 0.2733258 0.3380027 0.2227678
SWN  0.2858966 0.2633866 0.3650285 0.06478864 1.00000000 0.0834507 0.1815691 0.1082383
SJM  0.4897106 0.4780761 0.2233835 0.27332580 0.08345070 1.0000000 0.3131153 0.2069674
SLG  0.5973075 0.5664838 0.2819550 0.33800270 0.18156911 0.3131153 1.0000000 0.3146444
STJ  0.4858736 0.4396684 0.2501970 0.22276779 0.10823825 0.2069674 0.3146444 1.0000000
```

Figure 10-2. *Correlation matrix for entire data set*



```
Importance of components:
                          PC1    PC2    PC3     PC4     PC5     PC6     PC7     PC8
Standard deviation     1.9561 1.0354 0.8969 0.86337 0.80754 0.73743 0.57288 0.16610
Proportion of Variance 0.4783 0.1340 0.1006 0.09318 0.08151 0.06798 0.04102 0.00345
Cumulative Proportion  0.4783 0.6123 0.7129 0.80604 0.88755 0.95553 0.99655 1.00000
```

Figure 10-3. *Summary of principal components analysis (PCA) on data set*

In row 2 of Figure 10-3, you can see the proportion of the variance each principal component contributes to the data set. It must be stated for clarity that *principal components do not represent the features within the data set*. With that being said, we can consider principal component 1 to be a combination of features 1 through 8, PC 2 to be a combination of features 2 through 8, and so on. The general rule of thumb is to consider as insignificant principal components that contribute 1% or less to the total variance. When translating this to the data set, we would remove feature 8 within the data set. This same pattern of analysis should be extrapolated, but only when linear correlations between features are observed. Back in Figure 10-1, you can see generally moderate to strong linear correlations among the features, indicating that PCA is indeed an appropriate choice for features.

Model Evaluation

Now that we've preprocessed the data, let's consider our choices for algorithms. In this example, we'll evaluate a couple of different choices and evaluate the MSE on all of them. The number of models to choose is entirely up to you, but for this practical example I'll choose two. Furthermore, should you choose to evaluate statistics other than MSE, such as R Squared, it is reasonable to evaluate these measures relative to the goal of the experiment. That said, MSE should be and is the primary objective to minimize in regression models, and that should be the primary concern above all other evaluation methods.

Ridge Regression

Let's choose the first model: ridge regression. Here, we'll evaluate the MSE with respect to the value of the tuning parameter. In the following code, we begin by randomly sampling values from a normal distribution (5). These values will be used to pick the size of the tuning parameter, which we represent with K. The intuition behind this is that we'll sort the values from lowest to greatest and then compare the performance of our ridge regression model's MSEs by visualizing the error as we increase the tuning parameter:

```
#Ridge Regression
k <- sort(rnorm(100))(5)
```

In the following code, we begin by cross validating our results so that we are evaluating generalities of model performance rather than testing our algorithm on the exact same data set (6). We choose to use a training and test set of equal size, by splitting the data in half:

```
mse_ridge <- c()
for (j in 1:length(k)){ (6)
    valid_rows <- sample(1:(nrow(return_df)/2))
    valid_set <- new_returns[valid_rows, -1]
    valid_y <- new_returns[valid_rows, 1]
#Ridge Regression (7)
    ridgeReg <- lmridge(valid_y ~ valid_set[,1] + valid_set[,2] +
    valid_set[,3] + valid_set[,4]
                             + valid_set[,5] + valid_set[,6], data =
as.data.frame(valid_set), type = type,  K = k[j])
    mse_ridge <- append(rstats1.lmridge(ridgeReg)$mse, mse_ridge)
}
```

We then move to fitting the data to the ridge regression model using the `lmridge()` function and then append the MSE to a vector entitled `mse_ridge` (7).

When executing the following code, we see the result shown in Figure 10-4:

```
#Plots of MSE and R2 as Tuning Parameter Grows
plot(k, mse_ridge, main = "MSE over Tuning Parameter Size", xlab = "K",
ylab = "MSE", type = "l",
     col = "cadetblue")
```

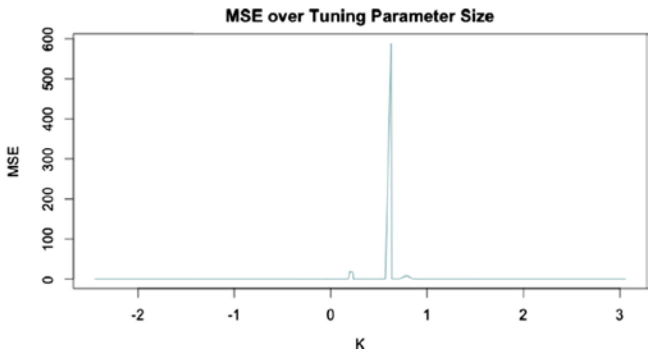


Figure 10-4. *MSE over tuning parameter size*

When looking at the plot, we see that the model performs best when our tuning parameter K is closest to the upper and lower bounds of the range displayed. Specifically, we'll choose to create a fitted model with a tuning parameter value of 1, as this K value yields a low MSE. When evaluating models it's important—in interviews, experiments, and for personal evaluation—to use plots to see the performance of the model with respect to some parameter value changing. This is useful for you as well as for other people who are using/evaluating your code. It will help to guide people through your thought process, and plots tend to be more engaging than looking at numerical outputs of code from a terminal.

Before we test our fitted model on data outside our validation set, let's show how we would tune another algorithm: the support vector regression (SVR).

Support Vector Regression (SVR)

The main parameter to tune here is the `kernel` function, which determines the shape of the hyperplane and therefore the shape of the regression line. When we execute the following code, we get the plot shown in Figure 10-5:

```
#Kernel Selection
svr_mse <- c()
k <- c("linear", "polynomial", "sigmoid")
for (i in 1:length(k)){
  valid_rows <- sample(1:(nrow(return_df)/2))
  valid_set <- new_returns[valid_rows, -1]
  valid_y <- new_returns[valid_rows, 1]

  SVR <- svm(valid_y ~ valid_set[,1] + valid_set[,2] + valid_set[,3] +
  valid_set[,4]
             + valid_set[,5] + valid_set[,6], kernel = k[i])
  svr_y <- predict(SVR, data = valid_set)
  svr_mse <- append(mse(valid_y, svr_y), svr_mse)
}

#Plots of MSE and R2 as Tuning Parameter Grows
plot(svr_mse, main = "MSE over Tuning Parameter Size", xlab = "K",
ylab = "MSE", type = "l",
       col = "cadetblue")
```

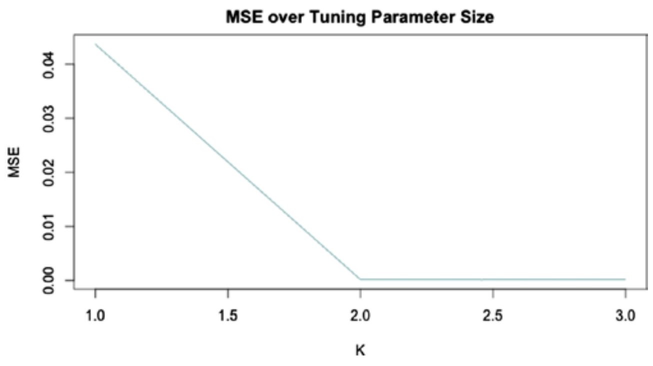
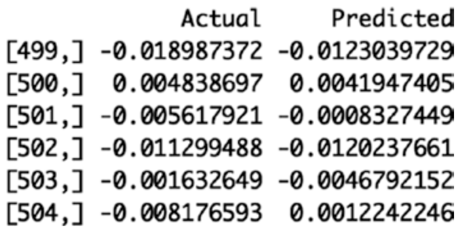


Figure 10-5. *SVR MSE with respect to kernel selection*

When evaluating the output, we notice that following MSE values in Figure 10-5. The polynomial kernel yields the smallest MSE and therefore is our choice. Now, that we've trained both models, we'll predict out of sample using our tuned models. In a practical setting, you should likely fit more than two models and evaluate the performance. Because this process is exhaustive, I've condensed this example to comparing two models for the sake of explanation. Regardless, let's see the performance of our tuned models:

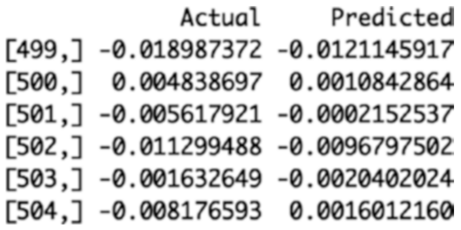
```
#Predicting out of Sample with Tuned Models
#Tuned Ridge Regression
ridgeReg <- lmridge(valid_y ~ valid_set[,1] + valid_set[,2] + valid_set[,3]
+ valid_set[,4]
                    + valid_set[,5] + valid_set[,6], data = as.data.
frame(valid_set), type = type,  K = 1)

y_h <- predict(ridgeReg, as.data.frame(new_returns[-valid_rows, -1]))
mse_ridge <- mse(new_returns[-valid_rows, 1], y_h)

#Tuned Support Vector Regression
svr <-   SVR <- svm(valid_y ~ valid_set[,1] + valid_set[,2] + valid_set[,3]
+ valid_set[,4]
                    + valid_set[,5] + valid_set[,6], kernel = "polynomial")
svr_y <- predict(svr, data = new_returns[-valid_rows, -1])
svr_mse <- mse(new_returns[-valid_rows, 1], svr_y)

#Tail of Predicted Value DataFrames
svr_pred <- cbind(new_returns[-valid_rows, 1], svr_y)
colnames(svr_pred) <- c("Actual", "Predicted")
tail(svr_pred)
ridge_pred <- cbind(new_returns[-valid_rows, 1], y_h)
colnames(ridge_pred) <- c("Actual", "Predicted")
tail(ridge_pred)
```

The preceding code uses the regression models we trained, except we set the parameter values based on which values yielded the lowest MSE. Although we fit the model to the training data, we're predicting on the test data. This is denoted by the fact that we're indexing from the return data frame using all the observations that we did not train the model against. When predicting on the test data set, the Figures 10-6 and 10-7 show the actual versus predicted stock values for each algorithm.



```
             Actual      Predicted
[499,] -0.018987372 -0.0123039729
[500,]  0.004838697  0.0041947405
[501,] -0.005617921 -0.0008327449
[502,] -0.011299488 -0.0120237661
[503,] -0.001632649 -0.0046792152
[504,] -0.008176593  0.0012242246
```

Figure 10-6. *Tail of actual versus predicted data frame (SVR)*



```
             Actual      Predicted
[499,] -0.018987372 -0.0121145917
[500,]  0.004838697  0.0010842864
[501,] -0.005617921 -0.0002152537
[502,] -0.011299488 -0.0096797502
[503,] -0.001632649 -0.0020402024
[504,] -0.008176593  0.0016012160
```

Figure 10-7. *Tail of actual versus predicted data frame (ridge regression)*

When evaluating the MSE of these algorithms, we receive the following results:

MSE for Support Vector Regression: 0.0002967161

MSE for Ridge Regression: 0.0002632815

Based on these results, it's reasonable to say that we should choose the ridge regression over the SVR based on the better MSE. You should feel free to work through the example given and use different feature selection algorithms, in addition to different algorithms altogether, when evaluating a solution. The purpose of this section, again, is to provide insight into how I generally approach these problems so that you may begin to develop your own methodology. Although there are general guidelines to model selection and tuning, everyone is free to perform this in their own way.

Let's now view a classification problem.

Problem 2: Speed Dating

In *speed dating*, participants meet many people, each for a few minutes, and then decide who they would like to see again. The data set we will be working with contains information on speed dating experiments conducted on graduate and professional students. Each person in the experiment met with 10–20 randomly selected people of the opposite sex (there were only heterosexual pairings) for four minutes each. After each speed date, each participant filled out a questionnaire about the other person. Our goal is to build a model to predict which pairs of daters want to meet each other again (that is, have a second date).

Problem Type: Classification

Any problem in which we're trying to determine binary or finite multinomial outcomes can be thought of as a classification problem. In this case, this will be a supervised problem, because we know the labels of the data beforehand, but we need to calculate them via a deterministic rule specific to this data set. A second date is only planned if both people in a given matching decide they would like to see the other person again. So, we'll create this column in the preprocessing stage of the data set:

```
#Upload Necessary Packages
require(ggplot2)
require(lattice)
require(nnet)
require(pROC)
require(ROCR)

#Clear the workspace
rm(list = ls())

#Upload the necessary data
data  <- read.csv("/Users/tawehbeysolow/Desktop/projectportfolio/
SpeedDating.csv", header = TRUE, stringsAsFactors = TRUE)

#Creating response label
second_date  <- matrix(nrow = nrow(data), ncol = 1)

for (i in 1:nrow(data)){
  if (data[i,1] + data[i,2] == 2){
    second_date[i]  <- 1
  } else {
    second_date[i]  <- 0
  }
}
```

As always, we begin the experiment by loading the necessary packages and clearing the workspace. Then we load the data and create a response label denoted second_date.

Now that we've gone through some initial preprocessing, let's describe and explore our data set. The features in this data set are as follows, from the first column through the last column:

- *Second_Date*: The response variable, y, for the data set which is binary. 1 = Yes (you would like to see the date again), 0 = No (you would not like to see the date again).
- *Decision*: The decision of the individual person, segregated by sex, as to whether they would like to go on a second date. 1 = Yes (you would like to see the date again), 0 = No (you would not like to see the date again).

- *Like*: Overall, how much do you like this person? (1 = not at all, 10 = like a lot).
- *PartnerYes*: How probable do you think it is that this person will say 'yes' for you? (1 = not probable, 10 = extremely probable).
- *Age*: Age.
- *Race*: Caucasian, Asian, Black, Latino, or Other.
- *Attractive*: Rate attractiveness of partner on a scale of 1–10 (1 = awful, 10 = great).
- *Sincere*: Rate sincerity of partner on a sale of 1–10 (1 = awful, 10 = great).
- *Fun*: Rate how fun partner is on a scale of 1–10 (1 = awful, 10 = great).
- *Ambitious*: Rate ambition of partner on a scale of 1–10 (1 = awful, 10 = great).
- *Shared Interest*: Rate the extent to which you share interests/ hobbies with partner on a scale of 1–10 (1 = awful, 10 = great).

Preprocessing: Data Cleaning and Imputation

Note that in this data set there are NA observations. As mentioned, we have multiple tools to deal with this problem, but it's important for us to algorithmically find a way to handle this. We will tackle that prior to performing any feature transformation. The following code shows the process by which we handle NA data:

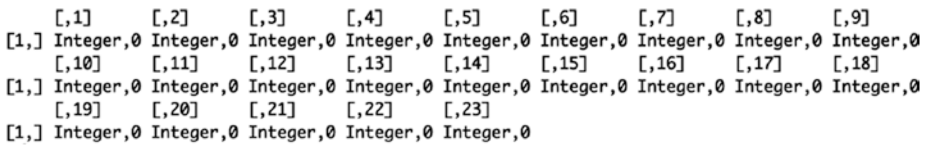
```
#Cleaning Data
#Finding NA Observations
lappend <- function (List, ...){
  List <- c(List, list(...))
  return(List)
}
na_index <- list()
for (i in 1:ncol(data)){
  na_index <- lappend(na_index, which(is.na(data[,i])))
}
```

First, we create a function that will let us append vectors to a list such that for each column, we have a vector of rows that indicate where the NA observations are. Given the nature of the data set, it's logical to impute the values using a method most reasonable given the data within that column/feature. Note that columns Second_Date, DecisionM, DecisionF, RaceM, and RaceF don't have any missing data. We're going to tackle the features that do have missing data.

We'll perform our data imputation using the expectation maximization (EM) algorithm described in Chapter 3. This is given in the amelia package, which can be installed from the R terminal. Before that, though, we must prepare our data slightly:

```
#Imputing NA Values where they are missing using EM Algorithm
#Step 1: Label Encoding Factor Variables to prepare for input to EM Algorithm
data$RaceM <- as.numeric(data$RaceM)
data$RaceF <- as.numeric(data$RaceF)

#Step 2: Inputting data to EM Algorithm
data <-  amelia(x = data, m = 1,  boot.type = "none")$imputations$imp1

#Proof of EM Imputation
na_index <- list()
for (i in 1:ncol(data)){
  na_index <- lappend(na_index, which(is.na(data[,i])))
}
na_index <- matrix(na_index, ncol = length(na_index), nrow = 1)
print(na_index)

 #Scaling Age Features using Gaussian Normalization
data$AgeM <- scale(data$AgeM)
data$AgeF <- scale(data$AgeF)
```

The EM algorithm can't handle *factors* (categorical variables). That means we must numerically encode these factors prior to their being inputted to the algorithm. After this, we execute the amelia function, which executes what we would like. Moving forward, we provide proof that there is no longer any NA data within this data set by indexing any NA values and then printing this output, yielding the result shown in Figure 10-8.



```
     [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]      [,9]
[1,] Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0
     [,10]     [,11]     [,12]     [,13]     [,14]     [,15]     [,16]     [,17]     [,18]
[1,] Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0 Integer,0
     [,19]     [,20]     [,21]     [,22]     [,23]
[1,] Integer,0 Integer,0 Integer,0 Integer,0 Integer,0
```

Figure 10-8. *Displaying counts of NA values in cleaned data set*

We've successfully removed all the NA observations and will perform the last bit of preprocessing before we move on to feature selection. Let's look at the distribution of ages with respect to both male and female. We code this as the following and receive the subsequent result:

```
#Scaling Age Features using Gaussian Normalization
summaryStatistics(data$AgeM)

Mean   Std.Dev Min Max Range
1 26.60727 3.509664  18  42    24
```

```
summaryStatistics(data$AgeF)

Mean   Std.Dev Min Max Range
1 26.24317 3.977411  19  55    36

#Making Histograms of Data
hist(data$AgeM, main = "Distribution of Age in Males", xlab = "Age",
ylab = "Frequency", col = "darkorange3")
hist(data$AgeF, main = "Distribution of Age in Females", xlab = "Age",
ylab = "Frequency", col = "firebrick1")
data$AgeM <- scale(data$AgeM)
data$AgeF <- scale(data$AgeF)
```

When visualizing the distributions of the data using the `hist()` function, the code yields the results shown in Figures 10-9 and 10-10.

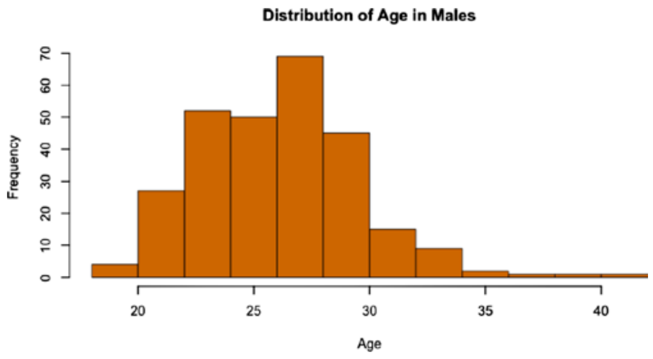


Figure 10-9. *Histogram of male ages*

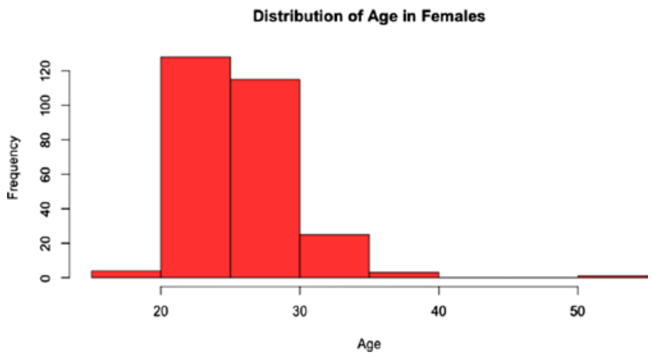


Figure 10-10. *Histogram of female ages*

The distributions of both female and male ages are positively skewed, meaning that the average is less than the median. However, note that there is significantly less variation in female ages in contrast to male ages. Although this also might serve as an insight we want to keep, you should glean the importance of displaying plots when exploring your data set and explaining what the information shows. This tends to be one of the most compelling ways to display information for people who aren't nearly as technical. For those who often find themselves making presentations, effective use of plots is a must. Finally, we end our data cleaning and preprocessing by performing Gaussian normalization on the age variables so that their inputs don't affect the accuracy of our classification models, because they are on different ranges than every other variable that isn't a numerical label.

Now that all the necessary preprocessing has been performed, we can approach the task of feature selection.

Feature Selection

This data set doesn't have an abnormally large number of observations, but 27 individual features likely makes for overkill and will unnecessarily weaken our machine learning algorithm's predictive power. As such, it is reasonable for us to eliminate unnecessary features, though we should be mindful of this process not necessarily being as straightforward as it appears.

When looking at the correlation matrix (the matrix is too large to be displayed here), we notice that there are generally weak to moderate linear correlations. We will likely be unable to get effective results from any models that rely heavily upon linear assumptions. When relating that to feature selection, we are similarly unlikely to get good results from using PCA. So, I chose to use a random forest to denote feature importance based on how much they affect the classification of an observation:

```
#Feature Selection
corr <- cor(data)

#Converting all Columns to Numeric prior to Input
for (i in 1:ncol(data)){
  data[,i] <- as.integer(data[,i])
}

#Random Forest Feature Selection Based on Importance of Classification
data$second_date <- as.factor(data$second_date)
featImport <- random.forest.importance(second_date ~., data = data,
importance.type = 1)
columns <- cutoff.k.percent(featImport, 0.4)
print(columns)
```

When executing the preceding code, the following columns are above the 0.4 threshold set for importance:

```
[1] "DecisionF"  "DecisionM"          "AttractiveM"    "FunF"    "LikeM"
[6] "LikeF"      "SharedInterestsF"   "AttractiveF"    "PartnerYesM"
```

These will be the features used in our training set, and we now can proceed to model training and evaluation.

Model Training and Evaluation

Now that we have a sufficiently reduced and transformed data set, it's time to go about the process of model selection. Because the function that determines the classification is not linear, we should look at functions that can handle this type of data. In the next problem, we'll use the following portfolio of algorithms:

- Logistic regression
- Bayesian classifier
- K-nearest neighbors

We'll tune each algorithm's parameters individually, evaluate the training set performance, and then predict out of sample. Once we've done this for all algorithms, we'll evaluate the results side by side and then choose the most optimal algorithm.

Method 1: Logistic Regression

It's suggested that when evaluating a portfolio classification algorithms you should always start with logistic regression. The reason is less because of the expectation for this to be the best algorithm, and more from the standpoint that this forms a baseline evaluation from which you can compare the different classification algorithms. In this experiment, we'll evaluate the performance of our models with respect to their AUC score, which is the area under the (ROC) curve:

```
#Method 1: Logistic Regression
lambda <- seq(.01, 1, .01)
AUC <- c()
for (i in 1:length(lambda)){
  rows <- sample(1:nrow(processedData), nrow(processedData)/2)
  logReg <- glm(as.factor(second_date[rows]) ~., data = processedData[rows, ],
  family = binomial(link = "logit"), method = "glm.fit")
  y_h <- ifelse(logReg$fitted.values >= lambda[i], 1, 0)
  AUC <- append(roc(y_h, as.numeric(second_date[-rows]))$auc, AUC)
}
```

We start by altering the threshold that determines whether we classify an observation as a 1 or 0 based on the lambda parameter. We iterate over the algorithm and append the AUC score based on this parameter to the AUC vector. After this loop of iterations, we should evaluate the performance visually by using a plot. When plotting the AUC score vector over the lambda value, we write the following code and observe the output shown in Figure 10-11:

```
#Summary Statistics and Various Plots
plot(lambda[-1], AUC, main = "AUC over Lambda Value \n(Logistic
Regression)",
     xlab = "Lambda", ylab = "AUC", type = "l", col = "cadetblue")
```

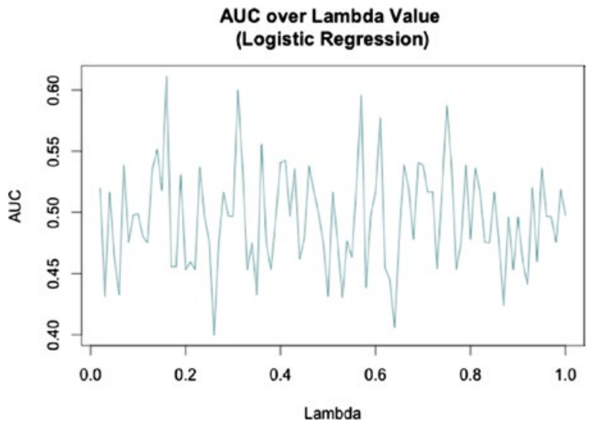


Figure 10-11. *AUC over lambda value*

We see that the AUC score is the highest when the lambda value is 0.15, so we'll use that lambda value. This is an example of how I would suggest you tune machine learning algorithms' parameters. Each parameter should be tuned individually so that you achieve a given objective, whether that is to minimize MSE or maximize AUC. In the logistic regression, the log odds threshold is really the only parameter we need to tune. We can view the performance of the tuned model over several iterations on the test set:

```
#Tuned Model
AUC <- c()
for (i in 1:length(lambda)){
  rows <- sample(1:nrow(processedData), nrow(processedData)/2)
  logReg <- glm(as.factor(second_date[rows]) ~., data = processedData[rows, ],
  family = binomial(link = "logit"), method = "glm.fit")
  y_h <- ifelse(logReg$fitted.values >= lambda[which(AUC == max(AUC))], 1, 0)
  AUC <- append(roc(y_h, as.numeric(second_date[-rows]))$auc, AUC)

}
```