Spring 2019

Regression Tree Construction for Reinforcement Learning Problems With a General Action Space

Anthony S. Bush Jr

REGRESSION TREE CONSTRUCTION FOR REINFORCEMENT LEARNING PROBLEMS WITH A GENERAL ACTION SPACE

by

ANTHONY BUSH

(Under the Direction of Stephen Carden)

ABSTRACT

Part of the implementation of Reinforcement Learning is constructing a regression of values against states and actions and using that regression model to optimize over actions for a given state. One such common regression technique is that of a decision tree; or in the case of continuous input, a regression tree. In such a case, we fix the states and optimize over actions; however, standard regression trees do not easily optimize over a subset of the input variables [2]. The technique we propose in this thesis is a hybrid of regression trees and kernel regression. First, a regression tree splits over state variables at a macro level, then kernel regression models the effects of actions with a smooth function at a micro level. Then non-linear optimization is used to optimize the kernel regressed function to find the best action and get a precise prediction of its value for any given state. This "best action" is then stored in the tree and is instantly retrieved upon making decisions. This is not only more appropriate for problems with continuous output, but also for problems with a discrete output since it also generalizes the knowledge over actions as well as states, providing for smarter decision-making. The capabilities of this technique are observed for a time series constructed to realistically model a stock problem.

INDEX WORDS: Time series analysis, Reinforcement learning, Machine learning, Q-learning, Regression trees, Kernel regression

2009 Mathematics Subject Classification: 15A15, 41A10

REGRESSION TREE CONSTRUCTION FOR REINFORCEMENT LEARNING PROBLEMS WITH A GENERAL ACTION SPACE

by

ANTHONY BUSH

B.S., Georgia Southern University, 2017

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

REGRESSION TREE CONSTRUCTION FOR REINFORCEMENT LEARNING PROBLEMS WITH A GENERAL ACTION SPACE

by

ANTHONY BUSH

Major Professor:	Stephen Carden
Committee:	Ionut Iacob
	Scott Kersey
	Divine Wanduku

Electronic Version Approved:
May 2019

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF SYMBOLS

Itemize environment:

- $\mathbb{R}$ Real Numbers
- $\mathbb{N}$ Natural Numbers
- E[X]=$\sum_{i=1}^{n} x_i P(X = x_i)$
- $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$
- MSE = Mean Squared Error = $\frac{1}{n} \sum_{i=1}^{n} (y_i - \bar{y}_i)^2$
- SSE = Sum of Squared Errors = $\sum_{i=1}^{n} (y_i - \bar{y}_i)^2$

CHAPTER 1

INTRODUCTION

As one can imagine, learning from investing in stocks is a complex task. Fitted Q-Iteration is one powerful method designed to handle decision problems of such complexity. To execute Fitted Q-Iteration, the problem must be cast as a Markov Decision Process which maps a state and action at a given index to a reward. The goal here is to find an optimal policy that maps a given state to the action that gives the highest reward. Fitted Q-Iteration uses a function "Q" that takes into account the reward of different actions and iteratively makes guesses and learns at each step to update the Q function until an optimal policy has been reached. The learning that Fitted Q-Iteration does is dependent on some form of machine learning algorithm which uses the state and action variables as inputs and maps them to an expected value. This can be done using a wide variety of different algorithms, but for this problem, we explore the capabilities of regression trees, especially the capabilities of regression trees under the proposed construction of splitting only on state variables and then using a more precise technique to select an optimal action.

Regression trees learn by splitting up the data iteratively until the data has been partitioned into fine enough parts that are homogeneous with respect to the output variable. In particular, the goal for constructing a regression tree is to make splits such that for every split, the two sets created from that split are the most different from each other and are each optimally self similar. Once the data has been partitioned into nodes, each node is assigned the average response variable as the output. As a result, the tree can be used to map inputs to an expected output. For Fitted Q-Iteration, the state and action together constitute an input which the tree then uses to map to a value indicating how desirable it is to be in that state, and choose that action. By optimizing over actions for each state, the policy is extracted.

Classically, tree algorithms consider a discrete action space and construct a different

tree for each action, at which point a prediction is made based on which tree yields the best result [6]. However for many problems we have a continuous or large discrete action space, so we need a different method to handle this type of input. As stated in the abstract, we resolve this issue by splitting data based on states and then once all the splitting that can be done on states is done, we optimize over actions using kernel regression to find a so called "best action".

Kernel Regression is a technique used to smooth noisy functions, or make smooth functions out of large amounts of scattered data. Making such a smooth function allows us to then use some non-linear optimization technique to converge to an estimate for a maximum of the data given. For this particular problem, stocks can be very noisy, and FQI only makes guesses, so what we are left with after the regression tree has split on state variables is scattered noisy observations of actions to values, so we apply kernel regression to create a smooth function out of all of this noise.

The only obstacle left now is that of how we construct data that accurately models that of an actual share price over time and is easy to replicate. One such discipline that deals with such models is that of Time Series Analysis. Not surprisingly, the task of analyzing stocks is one that has received constant attention ever since the idea of stock was conceptualized. Because of this, myriad techniques exist to attempt to accurately predict stocks. The field of Time Series Analysis amends its efforts to, for any process indexed by time (i.e. not just stocks), remove cyclic and long-term trend and observe the "stationary" residuals. A stationary time series can be thought of as a time series that doesn't change in covariance or mean throughout time. Removing cyclic, also called "seasonal", trends ensures that the mean does not alternate periodically. Removing long-term trends of growth or decay ensures that the mean does not grow or decay over time. Removing long-term changes in magnitude of fluctuations in data ensures that the covariance stays constant.

Once a time series has been reduced to stationary residuals, a time series analysis

approach selects a model to try to fit the data. Many different models exist that capture certain characteristics of the data being analyzed, and many are able to match data with surprising semblance [3] [10] [7].

Two such characteristics that are common in the stock market are mean reversion and shock. Mean reversion is the concept that eventually stocks return to their long-term mean. This concept can be rationalized using probability distributions. Since a probability distribution must integrate to 1, if a realized value is far away from the mean, then the number of values that are closer to the mean is higher than the number of values that are farther away from the mean. Stocks are shown to have this property when long-term and cyclic trends are removed [8]. In statistics, we refer to shock as "noise." Looking at any stock over time, it is easy to see persistent noise created by variability in trading and the fluctuating success of the company. This shock is not removed when long-term and cyclic trends are removed and is random, meaning that it doesn't affect stationarity. A process commonly used to model shock is the Moving Average Process which uses "white noise" to model the random shock over time. A process commonly used to model mean reversion is the Autoreressive Process which adds a percentage of the previous term or terms to create mean reversion. Both of these processes are combined to form the ARMA model which is a very commonly used method to model stock residuals. This model is described in more detail in section 2.5.

Since the ARMA model only models the stationary residuals of a stock, passive investment should on average yield only the mean with no profit. In other words, any trading strategy that can yield equity strongly above the mean in an ARMA time series would certainly do much better in a real stock where the seasonal and long-term trends were known (which would have to be known to construct the stationary model in the first place). For the proposed problem we attempt to form such a strategy.

CHAPTER 2

SUPPORTING METHODS

2.1 MARKOV DECISION PROCESSES

To introduce the concept of Fitted Q-Iteration it is necessary to lay the groundwork by discussing the idea of Markov Decision Processes. A Markov Decision Process is a Stochastic Process that models a situation where the next state is dependent on the current state and an action chosen by a controller or agent, after which a reward is received. As the name suggests, a Markov Decision Process has the Markov Property. That is, probabilities for the next state conditioned on the current state are equal to the probabilities for next state conditioned on not only the current state, but all other prior states as well. In other words, the probabilities for the next state only depend on the current state and action and not on any prior states.

To formalize, we begin by defining Stochastic Processes.

Definition 2.1. *A Stochastic Process is a sequence of random variables* $\{X_t\}$ *indexed by a set* T

Definition 2.2. *A Markov Decision Process is a Stochastic Process written as a 4-tuple,* $(S, A(s), P(s, s', a), R(s, a))$ *where* $X_t \in S$ *is the set of states,* $A(s)$ *is the set of actions available from state* $s \in S$*,* $P(s, s', a)$ *is the probability that the state* $s \in S$ *and action* $a \in A$ *will lead to state* $s' \in S$ *and* $R(s, a)$ *is the random variable reward for being in state* s *and using an action* a*. Note that elements of S and A can, themselves, be tuples.*

The goal of Markov Decision Processes is to find an optimal policy denoted $\pi^* : S \rightarrow A$ that maps a given state s to an action a returning the highest reward. The optimal policy can be thought of as the best choice to make in a given situation. To attempt to solve for an optimal policy, it seems natural to consider a value function $V^\pi : S \rightarrow \mathbb{R}$ that assigns a

value to state s for using policy π. A common definition of the value function $V^{\pi}(s)$ is the total expected discounted reward criteria [2].

$$V^{\pi}(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)|s_0 = s\Big] \tag{2.1}$$

Where $\gamma \in [0, 1]$ is the discount factor. γ serves two purposes. It corresponds to the idea that present rewards are more valuable than future rewards, and ensures convergence when rewards are bounded.

A simple rearrangement of the expected discounted reward criteria yields [11]

$$V^{\pi}(s) = E\Big[R(s, \pi(s))\Big] + \gamma \sum_{s' \in S} V^{\pi}(s') P(s'|s, \pi(s)) \tag{2.2}$$

This is known as Bellman's equation. Q-Learning is based on a generalization of Bellman's equation that includes actions. The expected discounted reward criteria in the form of Bellman's equation can be used to define for an optimal policy π^* by requiring $V^{\pi^*} = sup_{\pi} V^{\pi}(s)\ \forall s$. If transition probabilities and rewards are known, dynamic programming and stochastic optimization can be used to find an optimal policy [9], however if no such probabilities are known, but there are real or simulated data with observed rewards, then a different technique is required to approximate an optimal policy.

2.2 Q-Learning

Define a function $Q^{\pi} : S \times A \to \mathbb{R}$ that makes use of the definition of actions under the policy π. As such, we have

$$Q^{\pi}(s, a) = E[R(s, a)] + \gamma \sum_{s' \in S} V^{\pi} P(s, s', a) \tag{2.3}$$

where $V^{\pi}(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(s_t, a|s_0 = s\Big]$ is the expected discounted reward criteria and γ is the discount factor in $[0, 1]$.

If Q^{π^*}, the Q function under the policy π^*, can be determined or approximated on all of $S \times A$, then we can easily find an optimal policy by $\pi^*(s) = \text{argsup}_{a \in A} Q^{\pi^*}(s, a)$. The Q-Learning algorithm estimates Q^{π^*} with $\widehat{Q_t}(s, a)$ which converges to Q^{π^*} as t approaches ∞. For this algorithm, we begin with $\widehat{Q_0} = 0$, and for $t > 0$, use the recursive definition:

$$\widehat{Q_t}(s_t, a_t) = (1 - \alpha)\widehat{Q_{t-1}}(s_t, a_t) + \alpha\Big[r_t + \gamma \text{argmax}_{a \in A}\widehat{Q_{t-1}}(s_{t+1}, a)\Big] \tag{2.4}$$

where $\alpha \in [0, 1]$ is the learning rate. We include pseudocode for clarity in Algorithm 1.

Algorithm 1 Q-Learning Algorithm

$\widehat{Q_0}$=0, α=learning rate, M=number of iterations, γ=discount factor

for n in 1 to M **do**

s=observe current state

a=choose an action

u=state resulting from choosing action a

r= reward for performing action a

$\widehat{Q_n}(s, a) = (1 - \alpha)\widehat{Q_{n-1}}(s, a) + \alpha\Big(r + \gamma max_{b \in A(u)}\widehat{Q_{n-1}}(u, b)\Big)$

end for

For this technique, we can see that the algorithm learns at each step and then applies what it learns to select actions. This technique is known as online learning, and is very powerful since it continues to learn as more information is observed [2]. The only disadvantage, however, is that it is computationally inefficient. Learning can be slow since each observation is used once, and is then discarded. A technique that circumvents this issue is called, as one may guess, offline learning. In offline learning, the data collection and learning phases are separated. In the data collection phase, the actions are often chosen

at random. Then the algorithm learns from the collected data and updates Q. This offline variant of Q-Learning is known as Fitted Q Iteration.

Fitted Q Iteration can be much faster than traditional Q-Learning because it uses each observation multiple times, and uses all observations simultaneously [2]. For traditional Q-Learning, the algorithm makes an educated guess and then learns in the same step for each sequential step, meaning that they cannot be run in parallel. However Fitted Q Iteration can run in parallel and is therefore potentially more efficient. A disadvantage of Fitted Q Iteration is that it may not accurately predict the true maximum if the reward space is not fully represented by the observed rewards from random actions, but with large data sets, it is very likely that the true characteristics of the reward function can be extrapolated.

For the specifics of how Fitted Q Iteration is implemented, we begin with data in the form of four-tuples (s_t, a_t, u_t, r_t) where s_t is the state at time t, a_t is the (usually randomly chosen) action at time t, u_t is the next state at time t (i.e. $u_t = s_{t+1}$), and r_t is the reward of choosing action a_t while in state s_t. For the purpose of the algorithm, we initialize i_t to be (s_t, a_t) and $w_{0,t}$ to be r_t at each time t. i_t and $w_{k,t}$ can be thought of as the input and output of the algorithm respectively. We also initialize $\widehat{Q_0} = 0$ to begin with. Then the sequential portion takes place by updating the output as

$$w_{k,t} = r_t + \gamma \widehat{Q_{k-1}}(u_t) \tag{2.5}$$

for all time t and then evaluating $\widehat{Q_k}$ using some regression technique on $\{i_t\}$ and $\{w_t\}$. This procedure is carried out for k until some stopping criterion is met. We see the pseudocode for this in Algorithm 4.

A natural question arises as to which regression technique is suitable for evaluating $\widehat{Q_k}$ on each iteration. For this, we introduce Regression Trees.

Algorithm 2 Fitted Q Iteration

γ=discount factor, $\widehat{Q_0}$=0, k=1, T=set of observations, M=size of T, RM=regression model,

while stopping criterion not met **do**

for t in 1 to M **do**

$\mathrm{i}_t = (s_t, a_t)$

$w_{k,t} = r_t + \gamma \widehat{Q_{k-1}}$

end for

$\widehat{Q_k} = RM(i, w)$

k=k+1

end while

2.3 Regression Trees

Regression Tree Learning is a type of Machine Learning algorithm that trains on a data set of inputs and outputs and recursively partitions data into left and right subsets at several "nodes" to make a tree structure, which can be followed from the root node to return a predicted output for any input. The process for doing this is as follows:

1. Begin at node 1
2. Loop over input variables and values on which to bisect the data such that each part is maximally self-similar
3. Go node by node and repeat process until the observations in each node are sufficiently homogeneous in the output.

For Regression Trees, there are several key terms that come along with the implementation:

Algorithm 3 Regression Tree Construction

node=1, numnodes=1 T=set of observations, numin=number of input variables

while node $\leq$ numnodes **do**

If (node is splittable) do (

bestsplit=initial bad split

for i in 1 to numin **do**

splitcand = best numerical split for variable i

If (splitcand is better than bestsplit) do (bestsplit=splitcand)

end for

Create two new nodes labelled node+1 and node+2 and designate observations to each using bestsplit

numnodes=numnodes+2

node=node+1

)

end while

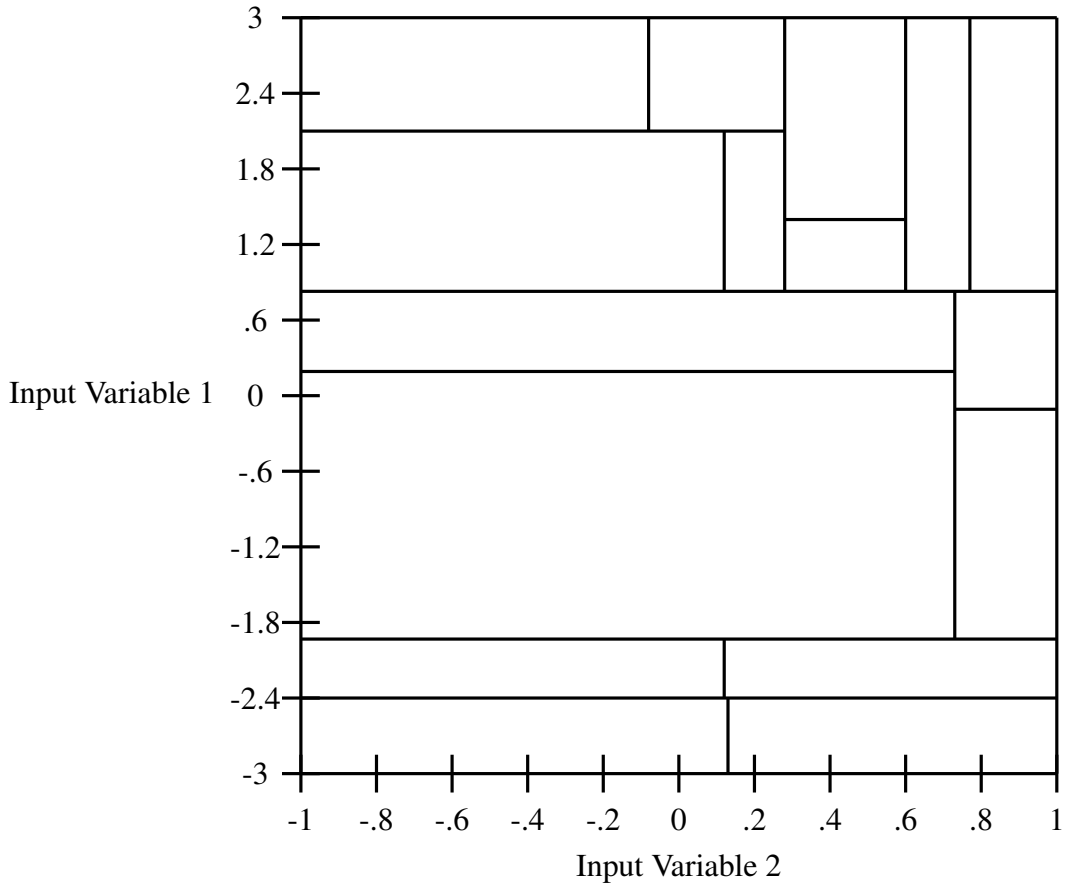

Figure 2.1: In this figure, we can see an input space partitioned by splits on Input 1 and Input 2.

- Each time a node splits, the two new nodes that are created are called “children” nodes
- The node from which these two children nodes originated is called the “parent” node
- When a node cannot split it is called a “terminal” node

To visualize, in Figures 2.1 and 2.2 we can see an example of how successive splits can partition an input space of two input variables to make a tree that makes predictions based on these splits.

As stated, the splits are determined in a way such that each “child” node contains data

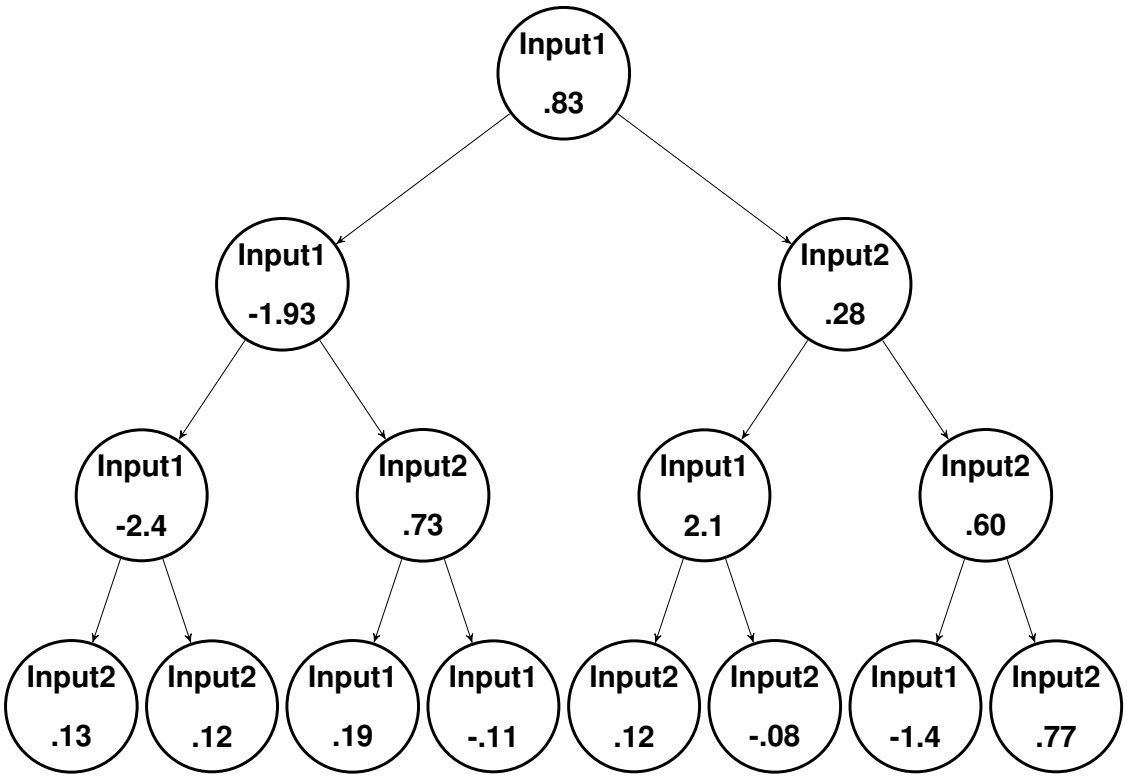

Figure 2.2: In this figure, we can see the tree that corresponds to the partitioned input space in Figure 2.1. We can see that there are 14 splits, 4 levels and 8 terminal nodes in this tree.

that is optimally self-similar. This is done so that the tree accurately decides where to send observations that display distinct characteristics for prediction purposes. The measure of how self-similar a node is most commonly calculated as some form of error. One way to make this calculation is using the SSE (sum of squares) for each group. In such a case, our goal is to minimize this error so that we have made a split that maximizes the homogeneity of both the groups. The formula for the error that we would like to minimize would then be the sum of the SSE's of both groups, stated below:

$$\text{Error} = \text{SSL} + \text{SSR} = \sum_{i=1}^{n_l}(y_i - \bar{y_l})^2 + \sum_{i=n_l+1}^{n}(y_i - \bar{y_r})^2 \tag{2.6}$$

where $\bar{y_l}$ denotes the average of observations of the left child, $\bar{y_r}$ denotes the average of observations of the right child, n denotes the total number of observations in the parent node, and n_l denotes the number of nodes designated to the left child under the proposed split. Later we will use n_r to denote the number of nodes designated to the right child under the proposed split.

Unfortunately, using (2.6) to calculate error requires looking at n observations for each split for each variable over each node. As one can imagine, this is a very costly calculation, and we would like a way to reduce its complexity.

There is a well known [13] simplification of SSE that goes as follows:

$$\textstyle\sum_{i=1}^{n}(y_i - \bar{y_l})^2 = \sum_{i=1}^{n} y_i^2 - 2y_i\bar{y} - \bar{y}^2 = \sum_{i=1}^{n} y_i^2 - 2\bar{y}\sum_{i=1}^{n} y_i + \sum_{i=1}^{n} \bar{y}^2$$

$$\textstyle= \sum_{i=1}^{n} y_i^2 - 2n\bar{y}^2 + n\bar{y}^2 = \sum_{i=1}^{n} y_i^2 - n\bar{y}^2$$

This is commonly referred to as the computational formula for the sum of squared errors and allows us to simplify (2.6) as

$$\textstyle\sum_{i=1}^{n_l}(y_i - \bar{y_l})^2 + \sum_{i=n_l+1}^{n}(y_i - \bar{y_r})^2 = \sum_{i=1}^{n_l} y_i^2 - n_l\bar{y_l}^2 + \sum_{i=n_l+1}^{n} y_i^2 - n_r\bar{y_r}^2$$

$$\textstyle= \sum_{i=1}^{n} y_i^2 - n_l\bar{y_l}^2 - n_r\bar{y_r}^2 = \sum_{i=1}^{n} y_i^2 - \left[\frac{(\sum_{i=1}^{n_l} y_i)^2}{n_l} + \frac{(\sum_{i=n_l+1}^{n} y_i)^2}{n_r}\right]$$

We notice here that $\sum_{i=1}^{n} y_i^2$ does not depend on which split we make, so in order to minimize (2.6) we need only maximize the remaining portion of the above formula, which we call the "error reduction", formalized below:

$$\text{Error Reduction} = \frac{\left(\sum_{i=1}^{n_l} y_i\right)^2}{n_l} + \frac{\left(\sum_{i=n_l+1}^{n} y_i\right)^2}{n_r} \tag{2.7}$$

In such a case, our goal is to find the split that maximizes the error reduction, thus minimizing the error within both groups and maximizing the homogeneity in both groups, indicating the best split. If we define $SL = \sum_{i=1}^{n_l} y_i$ and $SR = \sum_{i=n_l+1}^{n} y_i$, then we can update (2.7) iteratively by examining each split left to right and, at each one, updating SL to be $SL+y_i$ and SR to be $SR-y_i$ while updating n_l to be n_l+1 for one gained observation and n_r to be n_r-1 for one lost observation. This gives us an iterative strategy for comparing errors of splits. In particular, we have only a small fixed number of operations to compute rather than a number that increases directly with n, thus greatly reducing the complexity of the algorithm.

Also notice that the decision to label a node as terminal is dependent on whether or not it has enough observations to be able to split on and whether the output values are homogeneous. If splits were able to continue without stopping criteria, eventually the tree would split into data merely being near each observed value over all input variables. This is called "overfitting" and causes a tree to not only be costly to build, but costly to use. Overfitting occurs when the input space has been split so finely that it has become just as "noisy" as the original data. This happens in particular when the stopping criteria parameters haven't been selected appropriately for the training data and therefore have allowed the splits to continue for too long. A tree like this is useless because it merely returns the output corresponding to the input that is given rather than making a prediction on the regression of the data [14]. For this reason, it is left up to the programmer to determine the stopping criteria to make sure the tree is of the proper size. Multiple stopping criteria exist

that can determine this, including the cross-validated error, the MSE, and the minimum number of observations designated to terminal nodes [14]. More specifics on the stopping criteria used for this particular construction are detailed in Section 3.2.

Regression Trees are very useful for making predictions when considering all input variables, however they are not as accurate when regressing the output against input variables in isolation. For such a task, other techniques exist.

2.4 Kernel Regression

With noisy functions, attempting to use a gradient ascent (descent) method to converge to a maximum (minimum) can often be an exercise of futility since it is very likely to ascend to one of the myriad misleading local maximums provided the method converges at all. For this reason it is often a good idea to use some sort of smoothing algorithm that can be used to calculate less volatile gradients and increase the likelihood of reaching an estimate that is closer to the true maximum. One such algorithm is that of kernel regression.

As its name suggests, kernel regression applies a kernel density function to weight observations based on how close they are to a given input value. From this, a weighted average is calculated. An important parameter for kernel regression is the bandwidth, which controls the width of the kernel. With a proper bandwidth selected, the number of points near an input value should be relatively small, making this a fairly efficient extrapolation method. Also, if the largest gap in input values is smaller than the bandwidth, and the kernel density function is twice differentiable on its support, then the kernel regressed function is also twice differentiable, allowing us to calculate gradients and Hessians on it, which are needed for most optimization methods. Note that when we mention a gap between two input values, we mean the norm of the distance between these to points in their input space.

The process for creating a kernel regression is as follows:

1. Pick an input value. Note that this does not have to be a value from the training set.

2. For each point within bandwidth distance from the chosen input value, plug distance from input value into kernel function.

3. This assigns a weight to the observation that corresponds to the chosen input value.

An example of a noisy function extrapolated by kernel regression can be seen in Figure 2.3

For the computation of the norm of the distance between two input values, we need only apply a norm that is appropriate for the dimension of the input. In most cases, the euclidean distance suffices. The norm is then plugged in to a chosen function $f(n)$ where n is the norm of the distance between the input and the point. For $f(n)$ to be able to calculate weights for each norm, it needs to be non-negative, symmetric, and non-increasing for n at least 0. Since n in this case is a norm, it always will be. Once we have an appropriate norm, we calculate the weighted average as:

$$\widehat{K}(x) = \frac{\sum_{i=1}^{n} f(||x - x_i||) * y_i}{\sum_{i=1}^{n} f(||x - x_i||)} \tag{2.8}$$

A natural first choice for the kernel density function would seem to be a standard normal since it meets these criteria, but this function has infinite support and has no roots, so using this kernel would require calculating the kernel for all observations, many of which would be very small and would not contribute much to the prediction. For these reasons we would like a function that has roots, meaning it has finite support, so for each starting input value, a small number of norms have to be calculated. Several functions exist that transform a normal to work around this problem or use polynomials, trig, absolute value or other elementary functions to mimic the shape of a normal distribution. One very commonly used function for the kernel is that of Epanechnikov's kernel [5] defined

$$f(n) = \frac{3}{4}(1 - n^2) \tag{2.9}$$

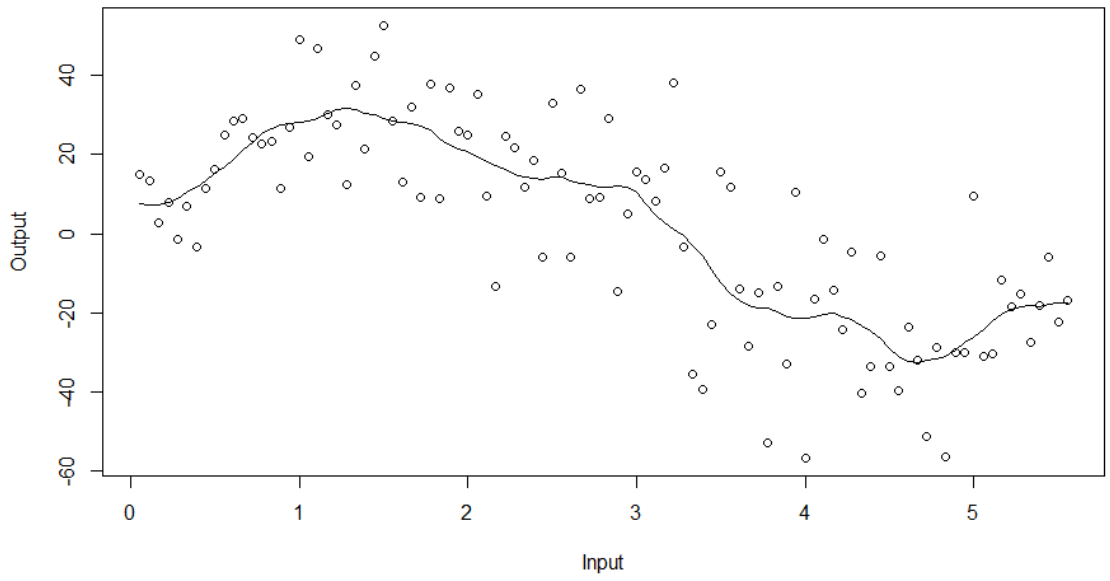

Figure 2.3: This figure is an example of a kernel regression of a sin function with noise. The kernel regression done here uses the $Kpred$ and $rKernel$ functions discussed in Section 3.3. The randomly generated data is marked in points, while the kernel regressed function is displayed as a curve. We can see that the height of the kernel regressed function at a given input value is determined by the outputs of input values multiplied by the kernel function of the norm of the distance between these input values. In such a way, the height of the kernel regressed function for a given input value is influenced more by observations with input values closer to the given input value.

Epanechnikov's Kernel is optimal in the sense that it minimizes the asymptotic mean integrated squared error or AMISE [12]. Also, since Epanechnikov's kernel is quadratic, it is twice differentiable on its support, making the kernel regressed function that results from using this kernel twice differentiable (provided the largest gap in input values is smaller than the bandwidth).

Epanechnikov's kernel has support $0 \leq n \leq 1$, but can be scaled to have a larger support by altering bandwidth. Bandwidth scales the kernel function for any function as

$$f_h(n) = \frac{1}{h} f(\frac{n}{h}) \tag{2.10}$$

where h is the bandwidth. (2.10) gives us a formula for Epanechnikov's kernel with bandwidth h

$$f_h(n) = \frac{3}{4h}(1 - (\frac{n}{h})^2) \tag{2.11}$$

We can see a picture of Epanechnikov's kernel in Figure 2.4.

As stated, the bandwidth is the width of the kernel. Choice of suitable bandwidth is left up to the programmer. Smaller bandwidths have the advantage of being more accurate provided they are not so small as to lead to overfitting. They also have a greater risk of reaching an input value that has no observations within the bandwidth resulting in false zero predictions which lead to false convergence or failure to compute gradients or Hessians. On the other hand, large bandwidths take longer to compute since they include more observations within the bandwidth that need to be added to the calculation of the prediction and can also lead to predictions being made outside of the domain of inputs. In addition, large bandwidths may also underfit, and miss the shape of the underlying function by making near-flat predictions thus losing the behavior of the function that need be optimized. An appropriate bandwidth depends on the data. If the data is sparse, it is best to use a large bandwidth, while if there is a large amount of data, and being far outside of the action

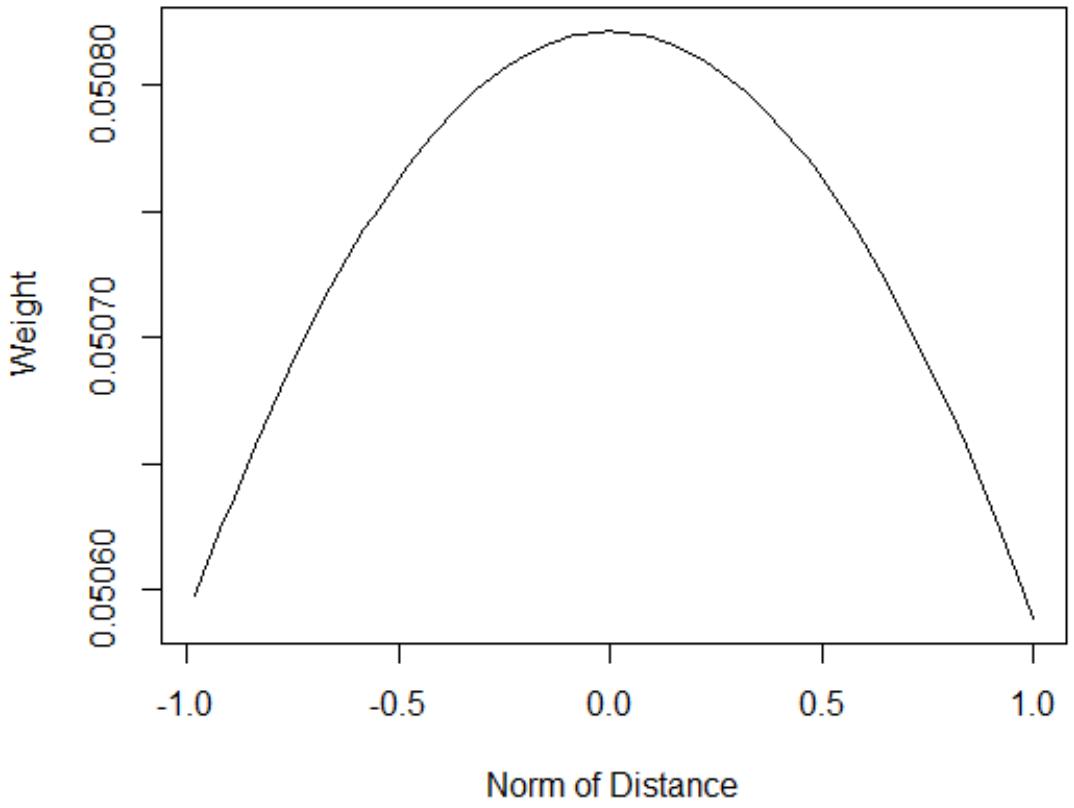

Figure 2.4: This is a graph of Epanechnikov's kernel with bandwidth 1. We can see that the closer the norm of the distance is to 0, the higher the weight is, meaning that the kernel gives more weight to values closer to a given input value. Note that the norm of the distance can never be negative, but we show negative values in this figure to give the reader an understanding of the shape of the kernel.

space is not an issue, it is best to go with a smaller bandwidth. Several studies have been conducted for optimal bandwidth selection. One of these is what is commonly referred to as the "Rule of Thumb" bandwidth [4]:

$$\hat{h} = \sqrt[5]{4(2 + c_v^2)} s n^{\frac{2}{5}} \tag{2.12}$$

Where s is the sample standard deviation, c_v is the coefficient of variation defined by $c_v = s/\bar{x}$, and n is the number of observations.

For data with multiple input variables, it is common for each variable to have a different mean and variance, so if a bandwidth is optimal for one variable in terms of (2.12), then it is not optimal for a variable with a different variance or standard deviation. Since (2.12) consists of standard deviation and average, we notice that if each input variable is standardized to have the same mean and standard deviation then the same bandwidth will apply to every variable. A natural choice then would be to have mean 0 and standard deviation 1, but a mean of 0 in (2.12) yields an undefined bandwidth, so we choose mean 10. This is done by, for each variable, subtracting each observation by the mean of the observations in that variable and dividing by the standard deviation of the observations in that variable, and then adding 10. Since standardizing the data automatically makes the mean 10 and standard deviation 1, this reduces the formula to:

$$\hat{h} = \sqrt[5]{8.04} n^{\frac{2}{5}} \tag{2.13}$$

As we can see, this reduced formula is only dependent on the number of observations in the data set. Note that it is important to "unstandardize" the inputs afterwards for the values to have any meaning. For this reason, when this method is applied, it is important to save the standard deviation and mean of observations for each variable before the standardization is done.

2.5 THE ARMA MODEL

As mentioned before, the ARMA Model is a stationary process that consists of a Moving Average Model and an Autoregressive Model. It is mean-reverting and "noisy," making it a good option for modeling the stationary residuals of stocks. We begin by discussing the concept of stationarity. For each one of the definitions in this section, let X_t be a sequence of random variables indexed by $t \in T$ such that T is countable.

Definition 2.3. *X_t is called stationary if $E[X_t]$ is independent of t $\forall t \in T$ and $Cov(X_t, X_{t'})$ is independent of t and t' for all $t, t' \in T$.*

Note that the definition used above is often referred to as "weakly" stationary, while "strongly" stationary refers to sequences such that X_t has the same joint distribution for every $t \in T$, but this definition does not usually apply to time series, and thus for the purposes of this problem, we will use the word stationary to mean weakly stationary, defined above.

Definition 2.4. *$\{X_t\}$ is called White Noise with mean zero and variance σ^2 if each random variable X_t is uncorrelated to one another and has mean zero and variance σ^2. We then denote $\{X_t\}$ as $WN(0, \sigma^2)$.*

Definition 2.5. *$\{X_t\}$ is called a First Order Autoregression or AR(1) Process if*

$$X_t = \phi X_{t-1} + Z_t \tag{2.14}$$

$\forall\ X_t \in \{X_t\}$ where Z_t is $WN(0, \sigma^2)$ and ϕ is some constant.

Note that a First Order Autoregression Process can be extended to a general Autoregression or $AR(p)$ Process by replacing ϕX_t with $\sum_{i=1}^{p} \phi_i X_{t-i}$ for constants $\phi_1, ..., \phi_p$, and forcing X_t to be stationary. This is done by making sure that all $\phi_1, ..., \phi_p$ are not on the unit circle. For $AR(1)$ Processes stationarity is achieved by simply making $|\phi| < 1$.

Definition 2.6. $\{X_t\}$ *is called a First Order Moving Average or MA(1) Process if*

$$X_t = Z_t + \theta Z_{t-1} \tag{2.15}$$

$\forall\ X_t \in \{X_t\}$ *where* Z_t *is* $WN(0, \sigma^2)$ *and* θ *is some constant.*

Note that a First Order Moving Average Process can be extended to a general Moving Average or $MA(q)$ Process by replacing θZ_{t-1} with $\sum_{i=1}^{q} \theta_i Z_{t-i}$ for constants $\theta_1, ..., \theta_q$. Since the Moving Average consists of only noise and a constant it is easy to see that it is stationary for any $q \geq 1$ and any choice of parameter(s) $\theta_1, ..., \theta_q$.

Definition 2.7. $\{X_t\}$ *is called an ARMA(1,1) Process if* $\{X_t\}$ *is stationary for all* t *and*

$$X_t = \phi X_{t-1} + Z_t + \theta Z_{t-1} \tag{2.16}$$

$\forall\ X_t \in \{X_t\}$ *where* Z_t *is* $WN(0, \sigma^2)$ *and* ϕ *and* θ *are constants such that* $|\phi| < 1$.

Note that an ARMA(1,1) Process can be extended to a general ARMA(p,q) Process by the same replacements for the extension of the Autoregression and Moving Average Processes, and add the requirement that the polynomials $(1 - \phi_1 x - \phi_2 x^2 - ... - \phi_p x^p)$ and $(1 + \theta_1 x + \theta_2 x^2 + ... + \theta_q x^q)$ have no common factors [1].

CHAPTER 3

PROCEDURES

Executing Q-Learning, constructing regression trees, and using Kernel Regression are all functions that are either built-in to R or available in R packages, however these functions restrict us to a traditional tree structure, whereas what we need to be able to do is only split on state variables and then use a different method for action variables. Likewise, what we need is an algorithm that can differentiate between state and action variables, store information on values and best actions, and return them for given state values. As a result, the scripts for Fitted Q Iteration, regression tree construction, and Kernel Regression have all been written from scratch. These programs have been designed to be able to work for state and action spaces of any dimension, whether continuous or discrete.

3.1 FITTING THE Q FUNCTION

For Fitted Q iteration, we use a technique similar to Algorithm 2 in section 2.3. We assign a discount factor γ called gam and a number of iterations called $iterations$ naming how many times $Qhat$ should update. Then we apply the same technique as Algorithm 2 from section 2.3, but we vectorize the computation of $w = r + gam * Qhat$ for speed and expand $Qhat = RM(i, w)$ where RM is a selected regression model, using a loop to make predictions using the tree found from $rtree$ over all values of t from 1 to $number\ of\ periods$* $number\ of\ trajectories$.

To make the predictions for the RM step, we introduce a function $treepred$ which takes $tree$ and $state$ as inputs and returns a $reward$ and $action$. $treepred$ does this by using $state$ to follow the tree all the way down to a terminal node and then returning the $bestaction$ for that terminal node in the tree. This whole process is done $iterations$ number of times until an optimal $Qhat$ is reached at which point we have constructed a tree that is an estimate for the best possible trading strategy for the modeled stock.

Algorithm 4 Fitted Q Iteration

γ=discount factor, $\widehat{Q_0}$=0, $k = 1$, T=set of observations (s,a,u,r), M=size of T, RM=regression model

while stopping criterion not met **do**

for t in 1 to M **do**

$i_t = (s_t, a_t)$

$w_t = r_t + \gamma \widehat{Q}_{k-1,t}$

end for

$tree = rtree(i, w)$

for t in 1 to M **do**

$\widehat{Q}_{k,t} = tree(u_t, w_t)$

end for

$k = k + 1$

end while

3.2 TREE CONSTRUCTION

To construct the tree that we use for the learning step of Fitted Q Iteration, we make a function $rtree$ which takes input values $input$, $output$, $numstate$, $minleaf$, tol, and returns $splitvar$, $splitval$, $children$, $parent$, $obsnode$, $bestactions$, $objective$, and h. These variables are defined here:

- $input$ is the input for FQI, i.e. the states and actions in matrix form with each column being a state or action variable and each row being an observation at a time step. If T is the set of observations in the form of (s, a, u, r), $input$ falls out of T as the (s, a) portion of T. As such, $input$ is a set of elements of $S \times A$

- $output$ is the output from fitted Q iteration, i.e. the estimates of the RHS of Bellman's equation (2.3). If T is the set of observations in the form of (s, a, u, r), $output$ falls out of T as the r portion of T. As such, $output$ is a set of elements of R

- $numstate$ is the number of states in the input. The number of actions in the input is easily calculable as the difference between the number of columns in the input and $numstate$, and referred to in several places as $numact$

- $minleaf$ is the minimum number of leaves allowed to be in a node

- tol, a tolerance, is the minimum MSE of the observations designated to a node to be able to consider a split on that node

- $splitvar$ is a long vector where each entry is the variable at which the node of that index splits. If a node does not split, that entry is 0

- $splitval$ is a long vector where each entry is the value at which the $splitvar$ splits the node of that index into left and right. Like $splitvar$, if a node does not split, that entry is 0

- $children$ is a 2 by long matrix such that each column represents the children of the node of the column index. The first row is the left child and the second row is the right child. If a node does not split, the first and second row are left with 0.

- $parent$ is the long vector where each entry is the parent of the node of that index. Node 1 has no parent, so $parent[1]$=0.

- $obsnode$ is a list of vectors, where each entry of each vector is the index of the observations from input and output that are designated to the node of the list entry index. e.g. the 5^{th} vector in $obsnode$ may be $[1, 3, 6, 8, 10]$ meaning that the observations designated to node 5 are given by rows 1,3,6,8,and 10 of the input and output.

- $bestactions$ is a $numact$+1 by long matrix where the entry in the first row is the node, and each other row represents a variable on which the best action has been found. e.g. the 5^{th} column may $[8, 4.5, 7.6]^T$. This means that for the 8^{th} node, which is the 5^{th} terminal node, the best action is to choose 4.5 for the first action variable and 7.6 for the second action variable. Note that unlike $splitvar$, $splitval$, $children$, and $parent$, the column index doesn't match up with the node, but rather the first row gives the node for which the best action is being found. This indexing is done for ease of looping over terminal nodes.

- $objective$ is the reward value for each column given by the terminal node and actions in the same column of $bestactions$. e.g. (in corroboration of $bestactions$ example) if the 5th column of $objective$ is 100, then the reward for reaching terminal node 8 and choosing 4.5 for the first action and 7.6 for the second action is 100.

- h is the bandwidth used in kernel regression for the observations designated to each terminal node. e.g.(in corroboration of $bestactions$ example) if the 5^{th} entry of h is 2.3, then the bandwidth used to do kernel regression on the observations designated

to the 5^{th} terminal node, node 8, is 2.3

Below, we can see the output returned from running $rtree$ on data randomly generated from a normal distribution:

```
> input=round(matrix(data=rnorm(100,mean=5),nrow=20,ncol=5)*10)/10
> output=round(rnorm(20,10)*10)/10
> minleaf=4
> tol=.003
> numstate=3
> input
      [,1] [,2] [,3] [,4] [,5]
 [1,]  4.7  4.9  5.1  4.5  4.7
 [2,]  3.9  3.5  3.8  5.9  3.5
 [3,]  4.4  4.9  6.1  4.5  6.4
 [4,]  3.7  5.7  3.4  5.6  3.2
 [5,]  5.1  4.0  5.3  5.7  5.3
 [6,]  4.5  4.8  4.6  5.2  5.3
 [7,]  4.6  6.0  4.4  4.7  5.7
 [8,]  6.0  4.2  5.2  4.7  2.9
 [9,]  4.3  6.1  3.9  6.0  5.2
[10,]  4.3  4.0  4.8  4.4  5.7
[11,]  4.9  6.7  6.5  3.7  5.0
[12,]  3.3  8.3  5.2  3.4  5.4
[13,]  5.5  5.0  5.7  4.8  5.1
[14,]  5.3  5.2  4.6  5.1  6.0
[15,]  5.4  3.7  3.1  4.3  4.6
[16,]  6.8  6.1  5.0  3.8  4.2
```

```
[17,]  4.4  5.2  4.4  6.1  4.5
[18,]  5.5  3.1  6.5  5.5  5.6
[19,]  4.3  5.8  3.9  5.1  4.1
[20,]  4.5  4.5  4.5  5.3  6.4
> output
 [1] 10.4  8.4 10.8  8.8  8.5  9.2  9.9  9.0 10.4  9.9 10.2  8.5 11.4 11
[17]  9.3 11.4 10.2  9.2
> tree=rtree(input, output, numstate, minleaf, tol)
> tree
$splitvar
[1] 3 2 0 0 1 0 0

$splitval
[1] 5.50 4.85   NA   NA 4.50   NA   NA

$children
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    2    4    0    0    6    0    0
[2,]    3    5    0    0    7    0    0

$parent
[1] 0 1 1 2 2 5 5

$obsnode
$obsnode[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
$obsnode[[2]]
 [1]  1  2  4  5  6  7  8  9 10 12 14 15 16 17 19 20

$obsnode[[3]]
[1]  3 11 13 18

$obsnode[[4]]
[1]  2  5  6  8 10 15 20

$obsnode[[5]]
[1]  1  4  7  9 12 14 16 17 19

$obsnode[[6]]
[1]  4  9 12 17 19

$obsnode[[7]]
[1]  1  7 14 16

$bestactions
     [,1] [,2] [,3] [,4]
[1,]  3.0  4.0  6.0  7.0
[2,]  3.7  5.3  6.1  5.1
[3,]  5.0  6.4  4.5  6.0

$objective
```

```
[1] 10.2  9.2  9.3 11.1

$h
[1] 0.8714194 0.6966458 0.7970090 0.8714194
```

The output variable from which it is easiest to visualize what the tree created looks like is the matrix $children$. From the output above, we can see that node 1 splits into nodes 2 and 3, node 2 splits into node 4 and 5, node 3 does not split etc. From this, one can easily draw a tree. As soon as the tree is drawn, we can also label each node and associate it with the variable and value that it splits on from $splitvar$ and $splitval$ respectively. The parents of each node are stored in $parent$ for traceback purposes and for checking the number of levels in a tree after it has been constructed. And, lastly $obsnode$ is output to detail the indices of observations designated to that node. These indices are mostly meaningless, but it can give us an idea of the size of each node.

The remaining outputs are the ones that have nothing to do with how the tree looks, but more so what happens after the tree has split on state variables only. This part of the $rtree$ function is detailed in the next section, but we can still interpret the results here. For the previously listed output, we can see that the first row of $bestactions$ is 3,4,6,7, meaning that these are the terminal nodes. This should be corroborated by seeing 0's in the 3^{rd}, 4^{th}, 6^{th}, and 7^{th} columns of $splitvar$ and $splitval$ should be 0's and NA's respectively, which we can see they are. We can also see from obsnode that these are the smallest nodes, which makes sense since, for reasons discussed later in this section, larger nodes have a higher likelihood of being able to be split. If we look at the 3^{rd} column of $bestaction$ we can see that if a state is partitioned by the tree to flow down to node 6, which is the 3^{rd} terminal node, then the best action would be to choose 6.1 for the first action and 4.5 for the second action. Furthermore, the 3^{rd} entry in $objective$ tells us that the kernel predicted reward of making that action for that state is 9.3. The 3^{rd} entry of h gives us the bandwidth used in

the kernel prediction of the standardized inputs designated to terminal node 6 at which the best action was determined.

Now we go more in depth into the details of how the tree is constructed. For our approach, we need to construct a tree that partitions over state variables only. Including all of $input$ and not just the state variables allows us to include the finding of the best action at each terminal node of the tree within the $rtree$ function. But before we do that, we need to make the tree. We do this by making splits until a certain stopping criterion is met. The way that we tell the tree to stop trying to split is that we define two variables: $node$ and $nextnode$. What we do now is to iterate by $node$ and attempt to make a split at each one. If we can't split (determined by several factors including $minleaf$ and tol), then we start over, but if we can split then we increase $nextnode$ by 2 for each new node created. Naturally, we continue this process until $node$ "catches up with" $nextnode$ i.e. when $node$ becomes the total number of nodes created.

For the actual splitting, it is stated in section 2.3 that the idea is to make splits such that the difference between the SSE of the observations in a node and the sum of the individual SSE's of the two child nodes is minimized. To do this with an iterative strategy, we make use of equation 2.7 from section 2.3 to make this calculation faster. Now, we have a way to calculate a split on a node, but first, it is important to consider if the node is even large and diverse enough to consider splitting and if so, which splits of that node are themselves worthy of consideration. Hence, we have several checks that we perform before splitting a node:

Check 1 Size of the node: To be able to effectively maintain $minleaf$ in two children nodes, the number of observations in the parent node must be at least 2*$minleaf$. If node has exactly 2*$minleaf$, there is only one possible split to consider, namely, right down the middle.

Check 2 MSE of the output values in the node: Checking the MSE of the observations desig-

nated to a node allows us to make sure that there is enough error to consider splitting on. If the observations designated to a node are already homogeneous enough, there is no need to make a split at that node.

Check 3 Variance of the input values in the node: If the variance of the observations in a node is 0, that means that every value in the node is the same.

Now that we have splitting criteria, we need to know for which values in each node it is even reasonable to split on. Surely we cannot split a node into two parts such that one part has less than $minleaf$, so the only splits we consider are splits that occur between the $minleaf^{th}$ to $n - minleaf^{th}$ observations. This ensures that each node that is created from a proposed split will contain at least $minleaf$ observations. Note that since we only consider splits from the $minleaf^{th}$ to $n - minleaf^{th}$ observations, we need the variance of the input values of just these observations to be > 0. So, Check 3 needs to be amendend to the variance of the input values of the $minleaf^{th}$ to $n - minleaf^{th}$ observations being > 0 as a requirement for splitting.

Upon creation of a split, $splitvar$, $splitval$, $children$, $parent$, and $obsnode$ are all updated to include information gained from this process. This is done until $node$=$nextnode$, completing the $splitvar$, $splitval$, $children$, $parent$, and $obsnode$ data structures, thus completing the construction of the tree over state variables. Now we need only find the best action to choose at each variable which constitutes $bestactions$, $objective$, and h, found in the function $findbestaction$ of the following section.

3.3 Finding the Best Action

Once the tree has been constructed by partitioning over state variables, the action variables we are left with mapped to the output forms a very noisy function. So, in order to maximize this function, it becomes necessary to smooth the function. The method we

use to do this is kernel regression. with a selected bandwidth. In order to implement kernel regression, we first need a function that transforms norms into weights for all observations within a given bandwidth. We define this function as $rKernel$ and use the Epanechnikov Kernel adjusted by bandwidth h to give us the function $\frac{3}{4h}(1 - (\frac{n}{h})^2)$. Once the weight is computed, we also need a function that can make predictions using the weights of nearby values. To make predictions on output for an input value say x, we calculate the weights of all nearby values and then, for observations near x, say $x_1, ..., x_n$, and observed outputs of those input values, $y_1, ..., y_n$, compute

$$\widehat{K}(x) = \frac{\sum_{i=1}^{n} rKernel(x_i, h) * y_i}{\sum_{i=1}^{n} rKernel(x_i, h)} \tag{3.1}$$

Note that we vectorize both of these functions in the program for computational speed. We define this function as $Kpred$ and is the smoothed function mapping actions to rewards. Now we need only maximize this function, which we do using $nlminb$, a non-linear optimizer on a bounded domain, in R. The $nlminb$ function requires a starting point, a function to optimize, an input, an output, a bandwidth, and an upper and lower bound. The reason we use $nlminb$ instead of simply nlm is because of this upper and lower bound. $nlminb$ makes sure that the predicted maximum is within these bounds so that we get a reasonable maximum. Since $Kpred$ is smooth, $nlminb$ easily converges to an estimate for a local maximum of this function. Note that $nlminb$ only finds minimums; however, we are looking for a maximum. To overcome this, we set the output as the negative output and minimize this. We then have a negative maximum, which we multiply by -1 to get a positive max.

To get the data for a terminal node ready to optimize using $Kpred$ and $nlminb$, we first do the standardization mentioned in section 2.5 for each variable, and then compute the Rule of Thumb bandwidth. Now, we know that $nlminb$ will converge easily, but to avoid getting a local maximum rather than the absolute maximum, we consider a grid of starting points for $nlminb$ across a possibly multidimensional plane of actions along with

several randomly picked observed values so that we are likely to avoid any astronomically inconveniently placed maximums that avoid the grid. Once the grid and random points are assigned, $nlminb$ is performed for each starting point and then the max of all of these is taken as the absolute max. All of this constitutes the function $findbestaction$ in the program and returns the $bestaction$, $objective$, and $bandwidth$ which are later placed in $bestactions$, $objective$, and h in $rtree$.

CHAPTER 4

RESULTS

4.1 TRANSITION-REWARD DATA CONSTRUCTION

To execute $QFit$, we need a model for data construction. This allows us to learn iteratively before we test as many times as we want. If we wanted to do this with real observed data, we would be limited as to the number of different ways that the time series would happen, and thus would not be able to learn as much. So we use an ARMA model to construct our data set, so that $QFit$ can learn from as many different possibilities as we would like. Since Q Learning uses Markov decision processes, where each state is only dependent on the previous state and action, using an ARMA process is appropriate for this type of learning.

First, we set up a configuration list that holds global variables and parameters. This list consists of $\phi, \theta, \sigma, number\ of\ periods$, $number\ of\ trajectories$, $minimum\ share\ price\ allowed$, and $maximum\ share\ price\ allowed$. ϕ and θ represent ϕ from the $AR(1)$ process and θ from the $MA(1)$ process. σ is the standard deviation of white noise, and the rest of the variables are self-explanatory. In the program, the configuration list is called $configList$. To construct transition-reward data, we need a function to take one step in time and assign a reward to a state and action, and a script to execute this over the desired number of periods and trajectories to create $trdata$, the matrix of transition-reward data. The function used to take one step in time and assign a reward to a state and action is called $transitionreward$. To construct $transitionreward$, we need a function that goes from a state S, and action A and transitions to a "next state" U. The variable S consists of $cash$, $share\ price$, $number\ of\ shares$, $white\ noise$, and $period$, also represented by t. Since we want the stock to follow an $ARMA(1,1)$ process, we calculate the $share\ price$ using the $ARMA(1,1)$ model. We remember from (2.16) in section 2.5 that the next term in an

ARMA(1,1) process is calculated

$$X_{t+1} = \phi X_t + Z_{t+1} + \theta Z_t \tag{4.1}$$

The variable A, representing action, is interpreted as the number of shares bought at a step in time so that all of the other variables in S are calculated following how they would be updated naturally given the action and $share\ price$.

$cash_{t+1} = cash_t - A_t * share\ price_t$

$number\ of\ shares_{t+1}$=$number\ of\ shares_t + A_t$

$white\ noise_{t+1}$ is assigned randomly from a normal distribution of mean 0 and standard deviation σ from $configList$

$share\ price_{t+1}$=$\phi * share\ price_t$+$white\ noise_t$+$\theta * white\ noise_t$ (if this value is above $maximum\ share\ price\ allowed$, then $share\ price_{t+1}$ is set to be $maximum\ share\ price\ allowed$ and vice versa for $minimum\ share\ price\ allowed$. This ensures that our share price is always between the $minimum\ share\ price\ allowed$ and $maximum\ share\ price\ allowed$)

Then the variable U is set to be the result of each of these executions in the same organization as S. i.e. U=$(cash_{t+1}, number\ of\ shares_{t+1}, share\ price_{t+1}, white\ noise_{t+1}, t+1)$. The reward R is naturally calculated as $R = cash_{t+1} + number\ of\ shares_{t+1} * share\ price_{t+1}$. The reward can be calculated at each time step, however in order to demonstrate the capabilities of this method, we hold off the reward calculation until the last time step, so that we are only looking at the final reward, and all of the actions that it took to get there.

To then construct $trdata$, we assign a starting state, and perform $transitionreward$ for each time step in $number\ of\ periods$ for the total $number\ of\ trajectories$. at each step , S is stored in the first 5 columns, A is stored in the 6th column, U is stored in the

7th-11th columns, and R is stored in the 21th column. The resulting $trdata$ is a $number\ of\ periods$*$number\ of\ trajectories$ by 12 matrix with each row being a variable, and each row representing observations at a time step. Note that under this construction, each trajectory comes after one another to avoid adding an unneeded dimension to our matrix.

4.2 Parameter Specifications

To decide which parameters we want for this experiment, we have a number of different selections to make. We begin by going over the parameters in a configuration list referred to as $configList$:

ϕ, θ, and σ are the parameters used in the $sharePrice$ which is modelled by an ARMA(1,1) process. For experimentation we will play around with these parameters to see if our tree structure can learn from different scenarios. The effect that each parameter has on the behavior of trajectories is described below:

ϕ is the parameter for the AR(1) portion of the ARMA(1,1) process. As we can see from equation 2.13 from section 2.5, ϕ affects the only part of this process that is not random, so a value of $\phi > 1$ makes the process increase over time, and $\phi > 1$ makes the process decrease over time.

θ is the parameter used in the MA(1) portion of the ARMA(1,1) process. More specifically, it controls how the observed white noise in the previous $sharePrice$ affects the current value for the $sharePrice$. A positive θ means that positive shock in the previous $sharePrice$ affects the current $sharePrice$ positively, while negative θ means that positive shock in the previous $sharePrice$ affects the current $sharePrice$ negatively. If θ is large in absolute value then the shock in the previous $sharePrice$ has a large effect on the $sharePrice$ whether positive or negative. σ is the standard deviation of the normal distribution from

which the white noise is drawn. Effectively, σ determines how much shock is in the process. Large σ means there is a lot of shock in the process, while small σ means there is little shock in the process. Notice that $\sigma = 0$ nullifies θ and eliminates any randomness in the process, making each trajectory deterministic, and only differing with respect to the chosen value for ϕ.

For the number of periods, or $endTime$, we need a parameter that will be large enough to show that $QFit$ can back-propagate. Without overdoing it, let us pick $endTime$=5

For $numTraj$ we want to choose a number of trajectories that our tree will have to learn from, so we want to choose a large number, $numTraj$=15000

For $priceMin$ and $priceMax$ we want to bound the stock so that fitted Q Iteration can fully explore the state action space quicker, so without giving any bias to one side or the other we pick $priceMin$=30 and $priceMax$=130 centered at 80

Aside from $configList$ we also want to set the conditions under which S will be initialized:

For the starting $cash$, we begin with an investment of $1,000$

For the starting $sharePrice$ we begin with a price of 80 as the center of $priceMin$ and $priceMax$

For the number of shares to begin with, we want to think about the beginning of this situation as having not bought any shares yet, so we start with $numShares$=0

For white noise, we don't want to give the model any positive bias, so that we can see that our tree is capable of obtaining a positive return. Therefore we choose initial $whtNoise$=0

For period, we simply choose a starting period of 1

To run $rtree$ we also need to choose $numstate$, $minleaf$, and tol:

For $numstate$, we select the number of state variables. Our state consists of the the five variables, $cash$, $sharePrice$, $numShares$, $whtNoise$, and per, so we pick $numstate$=5

For $minleaf$ we want to pick a minleaf small enough that it can make predictions on small enough clusters of data, but also not too small as to overfit the data. Also, the smaller $minleaf$ is, the more splits that will be created, resulting in the $rtree$ function taking longer with each iteration. So we pick $minleaf$=30 as a good middle ground.

tol is just used to make sure that we do not have an MSE too small on observations designated to a node. This is a check that should only be in effect in extreme circumstances, so we pick a small value of tol=100 which is small compared to an investment $1,000$

For running $QFit$, we also need to choose a value for γ and the number of iterations, $iterations$:

γ represents the lasting impact of rewards on $Qhat$, and is the way that $Qhat$ is able iteratively learn the effect that prior actions have on the end reward even if those actions don't immediately result in rewards themselves at that period. So in order to create a tree with a lot of foresight, we pick $gamma$=.9

Note that $iterations$ should be at least $number\ of\ periods$ from $configList$ so that $Qhat$ can use gam to propagate all the way back over each time step. So, for $iterations$, we set $iterations$ to go a little past $endTime$ and set $iterations$=$endTime$+1

4.3 Implementation and Analysis

As discussed in the previous section, ϕ, θ, and σ control the behavior of a set of trajectories, so we want to test out a few different combinations of these parameters to show that this tree structure is able to perform in various types of scenarios. So, we set up three different experiments to test out different combinations of these parameters.

4.3.1 Experiment 1: Realistic Stock Residuals

For the first experiment, we want to choose a combination that will yield stock residuals that we would expect under usual circumstances, so we pick ϕ=.999, θ=2, and σ=5, so that we have a fair amount of noise.

We create $trdata$ with these parameters using $transitionreward$ and get a process with a $sharePrice$ modeled by an ARMA(1,1) distribution. A representation of the $sharePrice$ over time t and trajectory can be seen in Figure 4.1.

Now, we run $QFit\ iterations$ number of times to build a tree that has fully propagated. This gives us a large tree with 26 levels and 3095 nodes in total. This tree is too big to represent here, however, we can look at the number of times the tree split on each state variable, giving us an idea of the importance of each state variable in the tree's ability to predict.

State Variable	Cash	Share Price	Number of Shares	White Noise	Period
# of Splits	364	691	112	376	4

Table 4.1: Here we can see the number of times the tree split on each state variable for data that resembles realistic stock residuals.

From Table 4.1, we can see that the most important variable for splitting was Share Price with 691 splits in total, followed by White Noise with 376. This is good, since this

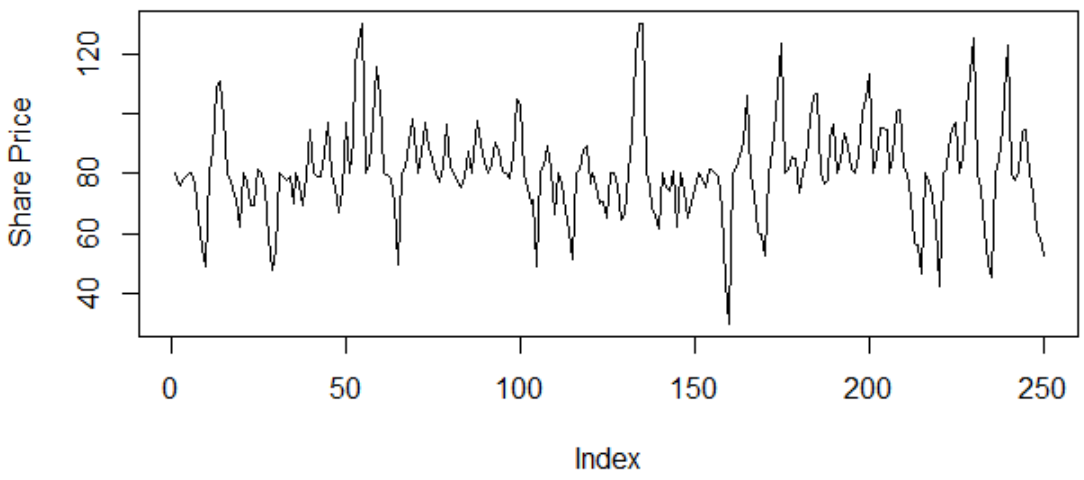

Figure 4.1: This is a plot of the share price over index in $trdata$. Since $endTime$ is 5, we show 50 trajectories, giving us an idea of how much the share price can vary over time and trajectory. As we can see, the trajectories are not biased above or below the initial share price of 80, so one would not expect to be able to make any profit above seasonal or cyclic trends from investing in a stock with such residuals.

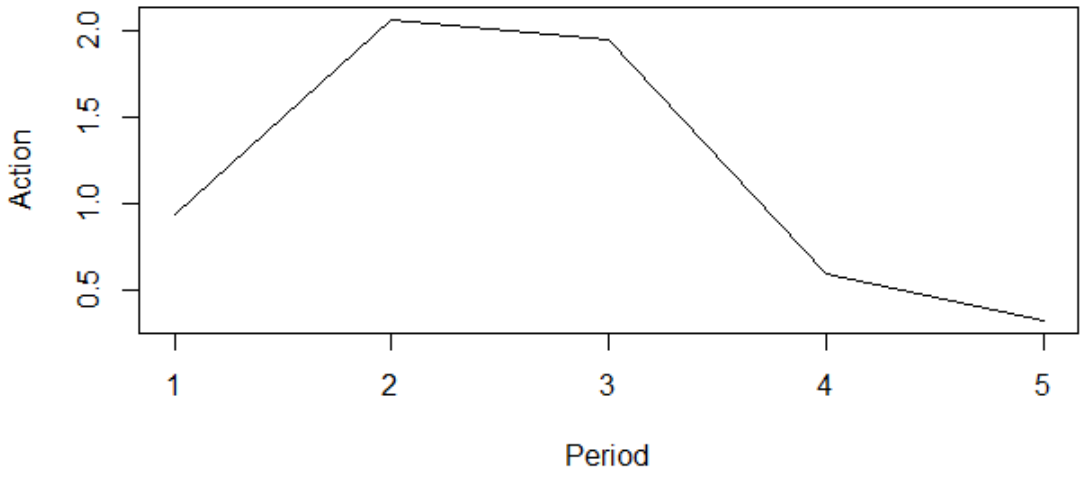

Figure 4.2: Here we see the average best action predicted by the tree over 10,000 trajectories. Note that with $sharePrice$ 80 and an investment of 1000, the most that our tree can buy at the first step is 12 after rounding to ensure a realistic action. We can see that under the parameters for this model, the tree usually buys the most during the early periods in each trajectory, thus reassuring us that $QFit$ has properly back-propagated. This provides us with a means for comparison to the tree's actions given different parameters.

means that our tree has learned to predict the next share price using the previous share price and white noise, which are both used in the calculation of the current share price.

Another aspect of the tree worthy of note is the average action that it takes at each period, shown in Figure 4.2.

To test the predictive capabilities of the tree, we test it against a random strategy where, at each time step, the action is chosen at random from the action space for that state, and a passive strategy which invests the full 1,000 for the initial action, and then chooses 0 for all subsequent actions. The random strategy represents a strategy that will yield the expected reward in a given scenario, and the passive strategy is a well known strategy that

many investors use along with diversification. Evaluating the mean of each strategy over 10,000 trajectories gives us Figure 4.2.

Strategy	Mean Reward	Standard Deviation
Tree	12.84828	1.894258
Passive	-0.5741673	1.740495
Random	-4.37676	3.268695

Table 4.2: Here we can see the mean reward and standard deviation of reward for each policy technique trading on realistic stock residuals.

As we can see from the mean reward column in Table 4.2, the tree policy does much better than passive or random strategies. With the small standard deviation observed, this means that the tree policy has yielded a strongly positive reward in a setting where it is not expected to be able to make a reward at all. This means that if we know the seasonal and long-term trend of a stock and the ARMA process that the residuals of a stock are modeled by, which are much easier to observe, then we can use this algorithm to learn from the ARMA model and earn a reward on top of cyclic and seasonal trend.

4.3.2 Experiment 2: Deterministic Bear Market Trajectories

For the second combination of parameters, we want to observe how this algorithm runs for a bear market with almost no noise, so we choose ϕ=.8, θ=2, and σ=.01. Having a ϕ of this size makes the process decrease for each trajectory. The goal here is to see whether or not the tree can pickup on declining trajectories.

Using these values to construct $trdata$ we observe trajectories each having a similar declining shape to the one in Figure 4.3. Given this strictly declining shape, we know that the best strategy would be to not invest anything, but does the tree learn to do that?

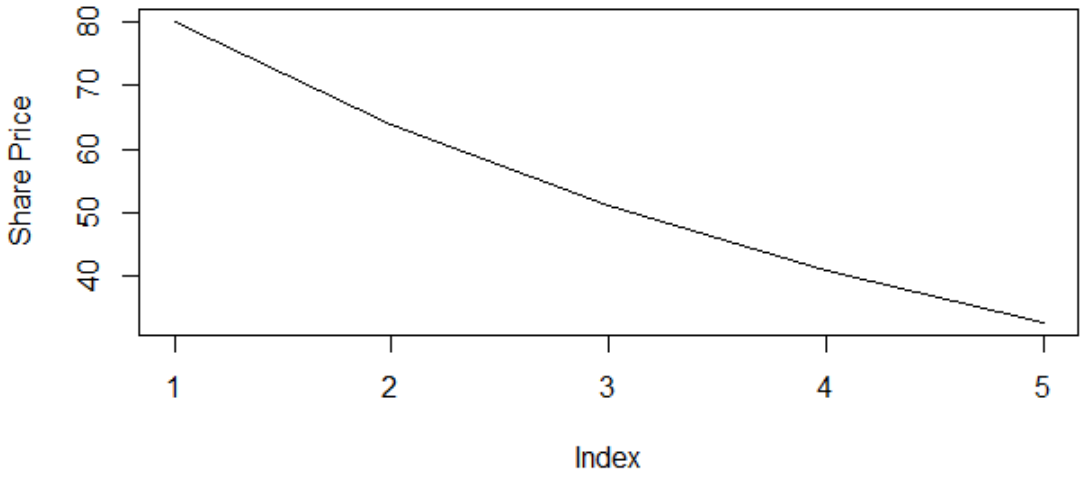

Figure 4.3: Here we see one trajectory of $trdata$ under these parameters. Since σ is so small, we can expect all trajectories created under these parameters to look almost identical.

Running $QFit$ we obtain a tree with 23 levels and 3029 nodes in total. Since we want to see if this tree is learning to invest nothing, we look at the average action for each period in Figure 4.4. From this we can see that, on average, the tree buys a lot at the start and then quickly switches to selling to avoid losing more money. This does not seem optimal, so is there a way to show that increasing the number of trajectories that the tree learns from will make the average action converge to zero? Making $numTraj = 1000$, much less than the previous $numTraj$ of 15,000, we get an average action for each period that looks like Figure 4.5.

Comparing, these two trajectories, we can see that the tree that learned from less trajectories seems to be more volatile in strategy with more propensity to buy than the more "experienced" one. From this, we can infer that the more trajectories we learn from, the closer to the optimal strategy we get. But how good is the first tree we looked at in this experiment with only 15,000 trajectories to learn from?

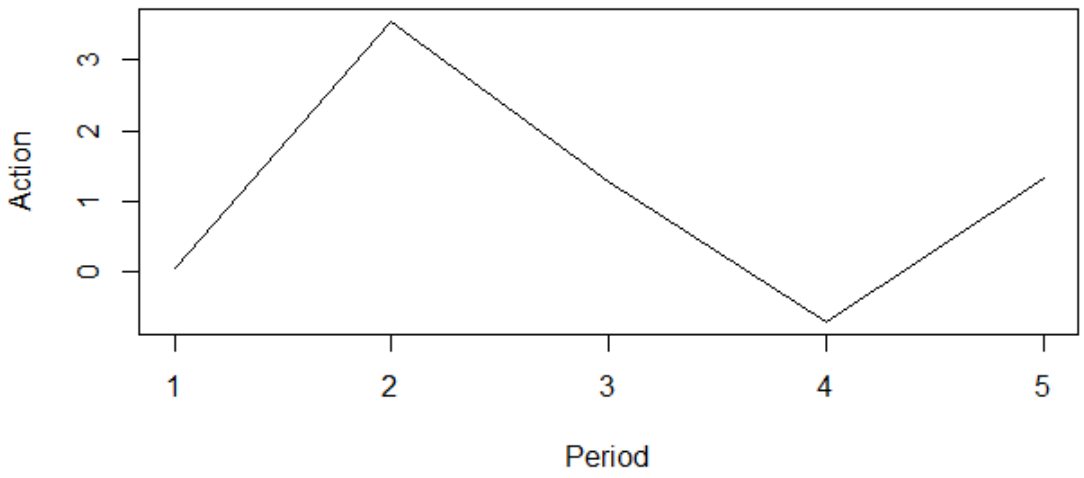

Figure 4.4: This is a plot of the average action over 10,000 trajectories chosen by a tree that has been trained on 15,000 trajectories. Here we can see that the tree chooses some investment at the start, and then quickly slows down its investment strategy.

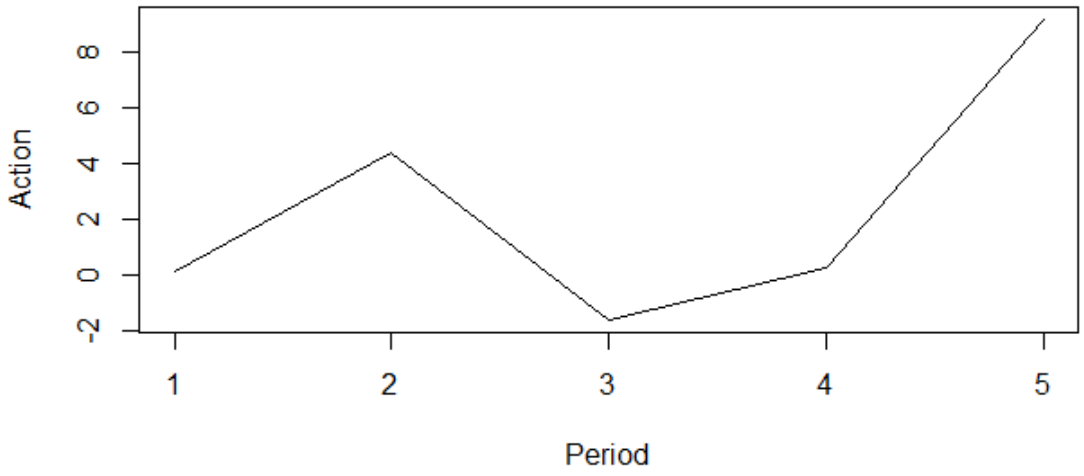

Figure 4.5: This is a plot of the average action at each period, 1 to $endTime$, over 10,000 trajectories. This suggests an average strategy of investing a middle-sized amount at the beginning, declining in buying amounts, and then finishing with a large amount of buying.

Strategy	Mean Reward	Standard Deviation
Tree	-142.6745	.510527
Passive	-602.7681	.0004110662
Random	-361.5742	.8612267

Table 4.3: Here we can see the mean reward and standard deviation of reward for each policy technique trading on near-deterministic bear market trajectories.

Comparing the tree to random and passive strategies we get:

Note that standard deviation is very small due to the near-deterministic trajectories brought on by low σ. From Table 4.3, we can see that, as expected, each strategy took a heavy loss. Passive strategy lost the most at around 603 which is to be expected since with this strategy, all of the user's value is declining for the entire trajectory. Random strategy does poorly as well which makes sense since one would expect to lose money no matter what, given a trajectory of this shape. We see that our tree strategy did poorly too, but not nearly as poorly as the other two losing less than half as much as random strategy. Hopefully, an investor who knew that a stock was modelled by such parameters would know not to invest anything at all in such a situation, but this experiment does a good job of depicting how a tree can learn from downward trends as well.

4.3.3 Experiment 3: Bull Market Trajectories

In contrast to the second combination of parameters. we want to observe how the algorithm will run for bull markets, so we choose ϕ=1.2, θ=5, and σ=2. $\phi > 1$ here makes the data increase over time. The purpose for this particular combination is to see if this algorithm can pick up on positive trend.

Creating $trdata$ with these parameters gives us upward shapes With $\sigma = 2$ and θ=2, these trajectories have a more noise than the trajectories of experiment 2, but not so much

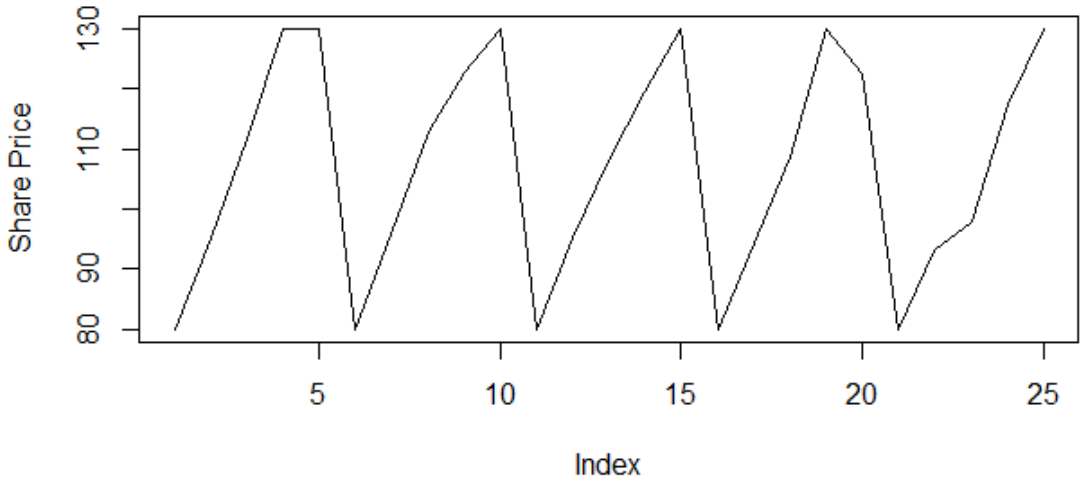

Figure 4.6: Here we can see five trajectories under the parameters detailed at the beginning of this experiment. As we can see, each trajectory is mostly increasing, but each one is somewhat different because of the noise in the process.

as experiment 1. We observe five trajectories with these parameters in Figure 4.6.

For such a case we know that passive strategy is a near-optimal strategy since each trajectory is mostly increasing. We say near-optimal because the optimal strategy would be to invest the maximum amount at every period, not just the first one. The reason so many investors use this method, along with diversification, is so that they can capitalize on the long-term upward trend in the stock market. For this example we hope to see the tree do better than random reward, but should not expect to exceed a passive strategy on average.

Running $QFit$, we get a tree with 29 levels and 3065 nodes in total. We also observe an average action for each period in Figure 4.7.

With an investment of 1,000 and an initial share price of 80, we can buy at most 12 stocks, rounding down to avoid buying fractions of stocks. We can see here that, over 10,000 trajectories, the tree on average immediately buys 11.8944422 shares amounting in

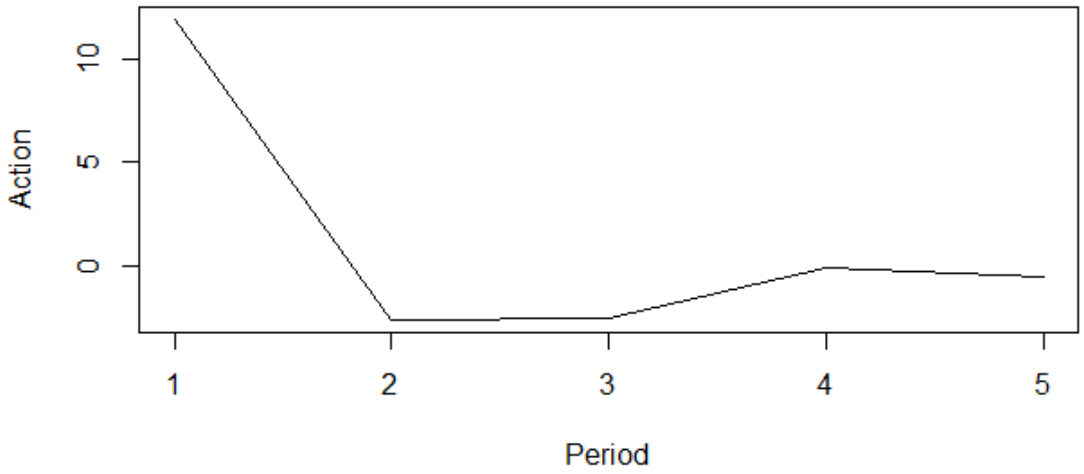

Figure 4.7: This figure is a plot of the average share price for each time step in 1 to $endTime$ over the course of 10,000 trajectories. From this plot, we can see that the tree begins by investing a large amount and then proceeding to invest small amounts over the remainder of the trajectory.

Strategy	Mean Reward	Standard Deviation
Tree	491.4343	.7567455
Passive	596.476	.3099254
Random	272.8962	1.141544

Table 4.4: Here we can see the mean reward and standard deviation of reward for each policy technique trading on somewhat random bull market trajectories.

roughly 951.56, nearly as much as the initial investment for passive strategy. We can see here that the tree is converging to an optimal policy. For a full comparison, we have:

As we can see from Table 4.4, tree strategy does strides better than random strategy, and still has a long way to go before converging to optimal strategy, but given the strategy that we observed, we expect that with a higher number of trajectories, we will converge to the optimal policy in this situation. In particular, we have seen that this tree structure can learn from yet another scenario.

CHAPTER 5

CONCLUSION

For reinforcement learning, we need to regress values against inputs in the form of states and actions and use that regression model to optimize over actions for a given state. When we use the regression model to optimize over actions for a given state we fix the states, and then find the most valuable action. For regression trees, fixing a subset of the input variables makes it very difficult to find values for the remaining input variables which maximize the output. So, we introduce a hybrid kernel regression tree that partitions the state space with a regression tree and then uses kernel regression to find the action that maximizes the reward for each part. To test the capabilities of this technique, we use the tree construction described to create an effective stock market trading strategy. As we can see, the strategy created by this construction has been proven to do much better than traditional trading strategies in realistic settings and to exceed expectations in unusual settings.

For training the kernel regression tree in these experiments, we chose $minleaf$ to be 30 and tol to be 100 to avoid overfitting; however, for many tree algorithms, cross validation is often performed to be able to get specific values for these parameters on individual data sets. With cross validation we are likely to converge to equal or better results with a lower number of trajectories.

REFERENCES

[1] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. Springer, 2002.

[2] Stephen Carden. Convergence of a reinforcement learning algorithm in continuous domains. Master's thesis, Clemson University, 2014.

[3] An-Sing Chen, Hung-Chou Chang, and Lee-Young Cheng. Time-varying variance scaling: Application of the fractionally integrated arma model. *The North American Journal of Economics and Finance*, 47:1–12, 1 2019.

[4] Su Chen. Optimal bandwidth selection for kernel density functionals estimation. *Journal of Probability and Statistics*, 2015.

[5] V. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications*, 14(1):153–158, 1969.

[6] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *J. Mach. Learn. Res.*, 6:503–556, December 2005.

[7] Massoud Metghalchi, Juri Marcucci, and Yung-Ho Chang. Are moving average trading rules profitable? evidence from the european stock markets. *Applied Economics*, 44(10-12):1539 – 1559, 2012.

[8] Alexander Michaelides and Yuxin Zhang. Stock market mean reversion and portfolio choice over the life cycle. *Journal of Financial Quantitative Analysis*, 53(3):1183 – 1209, 2017.

[9] S.M. Ross. *Applied Probability Models with Optimization Applications*. Dover Publications, 1992.

[10] Mohammad Mahdi Rounaghi and Farzaneh Nassir Zadeh. Investigation of market efficiency and financial stability between sp500 and london stock exchange: Monthly and yearly forecasting of time series stock returns using arma model. *Physica A*, 14(1):153–158, 1969.

[11] Juan Carlos Santamar ia, Richard S. Sutton, and Ram. Ashwin. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavor*, 1996.

[12] Stefanie Scheid. Introduction to kernel smoothing. 1 2004.

[13] Terry Therneau. User written splitting functions for rpart.

[14] Terry M. Therneau and Elizabeth J. Atkinson. An introduction to recursive partitioning using the rpart routines.