ALEX MORAN

# Predicting U.S. Monthly Inflation with Random Forest

I explore the process of creating and tuning a random forest model that can compete with classical time series models. The narrowly construed goal is to generate a univariate model that can outperform a standard ARIMA model when predicting on US monthly inflation data at a one-month forecast horizon. With some consideration of the unique features of time-series data, and of US inflation data in particular, it is possible to create such a model. Additionally, I begin to experiment with the random forest model at different horizons, as well as in a multivariate form.

Keywords: inflation, machine learning, random forest, time series.

Predicting the future is hard. It makes sense, then, to explore all of the resources at our disposal when we set out to forecast a time series. We should not limit ourselves to a certain class of models, as if only these could reasonably be expected to forecast well. The typical autoregressive (AR) and autoregressive integrated moving average (ARIMA) models are useful and valuable, both for their forecasting ability and for their specification of a time series’ underlying process. However, methods from machine learning such as the random forest offer an attractive alternative also to generate good forecasts, and their potential should be explored.

Given the rise of machine learning methods and their success in modeling cross-sectional data, and given the difficulty of forecasting time series data even with sophisticated classical models (described in the case of US inflation by Stock and Watson, 2007), it makes sense to consider applying some machine learning techniques to time series forecasting. Classical time-series models such as the ARIMA and the VAR are valuable, and they often perform significantly better than a naïve forecast. But machine learning is a valuable and underexplored resource when it comes to time-series data.

With this in mind, this paper offers a brief exploration of one particular machine learning technique, the random forest, and its success in forecasting a particular set of time series data, the US seasonally-adjusted monthly inflation rate. A standard random forest, of the type proposed by Ho in 1995, is meant for use with cross-sectional data; thus, this paper proposes a modification of the standard random forest in order to account for the peculiarities of time-series data as opposed to cross-sectional data. In the case of US monthly inflation, the modified random forest presents a sizable improvement over the standard random forest, both in terms of fit and in terms of forecast performance, as measured by root mean squared error (RMSE). Furthermore, the modified random forest presents fit and forecast results which compare favorably with those of an ARIMA model optimized by Akaike Information Criterion (AIC).

One drawback of the random forest is that, even given its success in fitting and predicting on a variety of datasets, the random forest as a model does not traditionally yield theoretical insight about those datasets. In this case, the random forest would not be expected to shed light on the motivating process for a time series, it does not answer the questions “why?” or “how?”; its success is limited strictly to the accuracy of its output. This paper challenges that convention by proposing a way to interpret the internal mechanism of the random forest in terms which speak to the changing state of the dataset over time. Furthermore, I can and do consider the reasons for the success of the random forest as a non-theoretical model. Why is it possible for a non-theoretical model to successfully forecast inflation?

The modified random forest described in this paper is univariate: it takes as input only lags of inflation. This random forest was designed with US monthly inflation at the one-month prediction horizon in mind, but I also consider its applications in a variety of contexts: as a multivariate model for US monthly inflation at the one-month horizon, as a univariate model for US monthly inflation across different prediction horizons, and as a univariate model for other types of time series: foreign inflation data and the domestic unemployment rate, for example. The model’s performance in each of these contexts sheds light on its success in the context of predicting US monthly inflation at the one-month horizon, and suggests how the model would need to be adjusted to best handle other types of time-series data.

The rest of the paper is organized in the following way. Section 1 describes the data used for the bulk of the paper, Section 2 describes the random forest as a concept in more detail, Section 3 describes the modifications made to the random forest in order to make it useful in time-series applications, Section 4 describes some results, and Section 5 looks ahead to further applications. Section 6 concludes.

1. THE DATA

The data which I consider for the bulk of this paper is a time series: United States Consumer Price Index (CPI) monthly inflation, seasonally adjusted, January 1959 – January 2020. I use the Consumer Price Index for All Urban Consumers dataset from the St. Louis Fed. Data is given at the monthly level. I produce inflation data by taking the log difference of monthly CPI data.

Time series data is tricky: it has characteristics that include seasonality, trends, and conditional heteroskedasticity, to name just a few. Many of the classical models (especially, the ARIMA model) are explicitly designed to handle those difficulties. The random forest method is not. It is designed for classification of cross-sectional data. With this in mind, I consider how to mitigate the challenging characteristics of time-series data.

To begin, I use seasonally adjusted data. This should account for seasonality issues. An Augmented Dickey-Fuller test rejects the null hypothesis of a unit root at the 5% level. In other words, there is very strong evidence that the series is stationary. A Spearman rank correlation test rejects the null hypothesis of no time trend and suggests the presence of a negative time trend in the data. A least squares regression of the data against a time trend and a constant suggests that time trend is significant at the 5% level.

The forecast period I consider for the bulk of the paper is January 1999 to January 2020. During this period in particular, there is limited evidence of a structural break in the series. Neither in the longer series from January 1959 to January 2020, nor in the shorter series from January 1989 to January 2020 is there evidence of a structural break, according to a supF test as described by Hansen in 1992. (As I will describe, the model only considers data from at most 8 years previous; thus, for the forecast period form January 1999 to January 2020, only data from January 1989 to January 2020 is relevant.) At 10-year intervals, beginning with the interval January 1989 – January 1999 and increasing by increments of one year, ending with January 2010 – January 2020, there is limited evidence of a few possible structural breaks.

For the random forest model I describe below, I use a dataset that includes the current month’s inflation , along with eleven lags on the current month’s inflation through , and a *trend* term, which has a value of 1 in January 1959 and increases with each month, finishing with a value of 733 in January 2020.

1. THE RANDOM FOREST

The random forest was first developed by Tin Kam Ho in 1995. It refers to a collection of regression trees, which are each trained on a randomly selected subsample of features (i.e., variables). This method allows the “forest” (the collection of trees) to grow increasingly accurate while also remaining resistant to overfitting as it grows in size. Ho’s work represented a major development in statistical modelling: “the accuracy increases with the addition of new trees” and yet “an increase in classifier complexity” did not lead to “overtraining” (Ho 1995). Thus, this method has the demonstrated potential to improve accuracy without the risk of overfitting—an ability which classical time series models lack.

*2.1 Objective Function and Splitting*

In Section 3, I customize the random forest for time-series applications. In the current section, however, I will merely describe the standard, “base” random forest that I will later customize. This base random forest is more simplistic than Ho’s. The forest is merely a collection of trees, each formed by randomized inputs to an identical pattern. The purpose of the regression tree is to sort a set of dissimilar data into more-similar subsets. To do this, the tree splits the data in such a way as to minimize the sum of squared errors, as shown in the equation below.

The equation above is the “objective function”; it stipulates the objective which, when achieved, will perform the optimal split of the data. The value of which satisfies the objective function indicates where the set of dissimilar data can be split into two subsets, with the minimum possible dissimilarity. In practice, the model operates as follows.

For each feature (i.e., independent variable) in a dataset, the model arranges the data in ascending order by that feature. The purpose of this sorting step is to eventually divide the dataset into two complementary subsets, each of which contains only consecutive values of the given feature. Once the dataset is sorted, the model identifies the exact observation at which a split will minimize the sum of squared errors for the data. In other words, the model computes what the sum of squared errors on the dependent variable *would* be, if the dataset were split into two complementary subsets, one of which containing all observations where values of the independent variable are equal to or less than a given value, and the other of which containing all observations where values of the independent variable are greater than that value.

Thus, each value of the independent variable is associated with a certain sum of squared errors value that *would* result, if the dataset were split at that value of the independent variable. There will be one possible sum of squared errors value which forms a minimum in the series, and there will be one value of the independent variable which is associated with this minimum. This value of the independent variable provides is the dataset’s optimal split point for that variable.

The model now records this optimal split point as a possible splitting point. It repeats this process for each feature in the data. Each feature is now associated with one possible split; of these, the one that results in the lowest sum of squared errors is the one that is truly optimal. The model splits the dataset at this truly optimal point, resulting in two complementary subsets, called “nodes.”

From the original dataset, two nodes are now produced. Within each resulting node, the model starts the same process over, splitting each node into two smaller nodes. This continues until some condition is reached that causes the model to stop splitting the data. For instance, the model could stipulate that each node must contain at least 10 observations; thus, if the model at some point produced a node with, say, 19 observations, it would not split that node any further (since a split would necessarily produce a node with less than 10 observations).

Each node, including each terminal node (i.e., “leaf”) is characterized by some splitting filter or series of splitting filters. If is the dependent variable, and if the independent variables are a *trend* term and through , then an example of such a series of filters may be

This is a node at the fourth level; it is the result of four prior splits. First, the tree split at the value of 0.00723…. Then, one of the resulting nodes was split at the value of 0.00560…, producing two smaller nodes. One of those two nodes was split based on the *trend* value, creating two yet smaller nodes. Finally, one of those was split at the value, creating the node referenced above.

All observations in the dataset which pass those filters are assigned to the same node, and that node will contain only those observations which satisfy the condition which characterizes the node. Note that is referred to twice in this sequence. This will often happen; each filter in the sequence is generated based on the data in the given node, independently of previous filters.

*2.2 Fitted Values*

Within each terminal node, the tree will compute the sample mean of all values. This sample mean will be the fitted value for each observation in that node, and therefore the predicted value for any observation in the test set which happens to satisfy the filters above. For each tree, the predicted value for the next period’s observation will be equal to the mean of the values of observations at the leaf which the next period’s observation *would* occupy (if the current period’s values were lagged, so that the current becomes in the next period, etc.). In the random forest, the predicted value for a new observation will depend on the predictions generated by each of the trees in the forest. Each tree will generate a single prediction for the next period value. The forest’s prediction will be the mean of those predictions.

* 1. *Parameters*

As mentioned above, the depth of each tree (i.e., the number of splits which generate a typical leaf) is generally limited by some externally imposed parameter. In the case of the “base” forest, I impose a minimum leaf size of 10 observations, in addition to a fractional penalty. The 10-observation limit seems to be a large enough minimum to avoid potential overfitting, but a small enough minimum (in a dataset of 733 observations) to allow for some diversity. The choice of a 10-observation minimum is admittedly arbitrary (why not 9 or 11?) but is reasonable for the “base” forest.

The minimum leaf size restricts the depth of the tree in conjunction with a penalty term. Each split in the tree is performed in order to minimize the sum of squared errors on the tree’s fit. A penalty requires that each split reduce the sum of squared errors *by a certain amount*, either by an absolute amount or by an amount relative to the previous sum of squared errors at the parent node (i.e., a fractional penalty).

The penalty I’ve imposed is fractional, with a value of 0.9. The penalty requires that each split ensure that the combined sum of squared errors within the two nodes that result from the split has a value of at most 0.90 of the original sum of squared errors within the original node. A penalty value of 0.75 would require each split to reduce the sum of squared errors by at least 25%, while a penalty of 0.99 would require each split to reduce the sum of squared errors by only 1%. As a result, a lower penalty term (i.e., a stricter penalty term) will result in a much shallower tree than a higher penalty term. The choice of penalty term will depend on the data and the model.

By itself, the 0.9 penalty term could yield a fairly deep tree with many terminal nodes, depending on the variability within the dataset. However, in conjunction with the 10-observation minimum, it will serve as a filter to prevent splits that are not meaningful (i.e., splits that don’t reduce the sum of squared errors by at least 10%), without causing the tree to go too deep.

The penalty term relates to the standalone tree function. Within the context of the broader forest, there is another parameter to consider: the feature fraction. In the forest, each tree does not consider each feature (variable) in the dataset; instead, each tree considers a random subsample of the features. The feature fraction parameter refers to the fraction of features that are used in creating each tree. In the base model, the feature fraction is 0.7, so each tree within the forest is created using a random sample of 0.7 (i.e., 70%) of features in the data set. A significant strength of the random forest method is its resistance to overtraining; the randomness which the feature fraction governs plays a key role in maintaining that strength.

Not only is each tree trained on a random sample of features, it is also trained on a random sample of the available data. In the base model, each tree receives a dataset which is sampled with replacement from the original data and which contains as many observations as the original data. Both these elements of randomness are supposed to prevent overfitting while preserving or improving accuracy.

I will stipulate that the “base” model contain a fixed number of trees: 50. Theoretically, there is no downside to adding more trees (although there will be diminishing positive returns in accuracy). But the random forest method is computationally heavy, and my resources are limited. Fifty trees should be enough to guarantee a reasonable level of accuracy.

1. REIMAGINING THE RANDOM FOREST

The random forest I’ve just described is not expressly designed to handle time-series data, but with a few adjustments, it could present an improvement over a classical ARIMA model, in terms of forecast accuracy. To realize that potential improvement is the goal of this project, and the adjustments I employ to that end are outlined in this section.

*3.1 Considering the Underlying Process*

The random forest is more or less agnostic about the process which generates its data. However, it seems reasonable to suppose that a time series like inflation is generated by some sort of autoregressive process. This intuition, if it is accurate, can be harnessed to make the forest better at predicting the future.

The autoregressive integrated moving average (ARIMA) model does a good job of predicting future inflation. The model is autoregressive in that it offers a regression on lags of itself; it is “integrated” in that it automatically differences the data (if necessary) in an effort to make the series stationary; and it is a moving average model in that it regresses on some proxy for prior residual terms. This model does more than just offer predictions about the future; it offers, in the abstract, a theory of what sort of process generates the data (an autoregressive moving average process); and it offers, in its implementation, a specific hypothesis about exactly which autoregressive moving average process generates a specific set of time-series data.

If the ARIMA model yields good predictions, then the model is not only useful as a sort of black box, which takes a certain data input and generates a prediction. Rather, the model is also useful in that it tells indicates *why* the data behaves the way it does. The predictions are good precisely because the model has correctly hypothesized, not only that the time series is generated by an autoregressive moving average process, but also that the autoregressive moving average process takes a particular form (e.g., it contains four AR terms and two MA terms, the coefficient on the first lag is 0.78 with a certain distribution, etc.).

Throughout the process of customizing the random forest model, it would be unwise to ignore the good information the ARIMA model provides. Instead, I intend to use every tool at my disposal to generate the best model possible. Now, for the data, I find that on a one-month forward prediction of US monthly inflation, from January 1999 to January 2020, the ARIMA model yields a root mean squared error (RMSE) of 0.0027, while a naïve model (in this case, the assumption that the value of inflation in a given month will be equal to that of the previous month’s inflation) yields an RMSE of 0.0031. This indicates that the ARIMA model does a better job of modelling the data than the naïve model does, and that the data is at least partially generated by the process which the ARIMA model describes.

Armed with this information, I no longer have to remain agnostic about the process which generates the data. Instead, as I construct the random forest model, I can assume that the data is at least partially generated by an autoregressive moving average process. In practice, it will be difficult to generate residuals in the random forest, so I will focus on the autoregressive part of the ARIMA model. (The “integrated” part is already taken care of, since the data is stationary.) In effect, I assume an AR process for the data.

The most basic AR process is an AR(1): an autoregression on only the first lag of a time series. Each observation in time is generated by some weight of the previous observation, plus some i.i.d. error. Since I know that the real data is at least partially generated by an AR process, it would be worthwhile to think about how the random forest would handle some artificial data that was generated from an AR(1) process. In other words, if I were presented with data which I knew with certainty was generated by an AR(1) process, how could I adjust the random forest to better model this data?

*3.2 Assessing Results with Simulated Data*

To answer this question, I first simulate a time series based on an AR(1) process with a coefficient of 0.9. Of course, inflation data is *not* a perfect AR(1) series, but adjusting the model on the assumption that it is will be a good first step to improving the accuracy of the model. The time series I generate contains 511 observations. I then assess the fit given by an ARIMA model optimized by AIC with the fit given by the base tree described in Section 1. The fits are roughly similar; the ARIMA has an RMSE of 1.016, while the base tree has an RMSE of 1.029. The ARIMA fits the data slightly better than the tree does, but the tree still provides a fairly comparable fit.

However, I ultimately want a random forest model which will forecast well, not just one that will fit well. Therefore, using first the ARIMA optimized by AIC, and then the base tree, I predict one period ahead for each of the last 100 observations (i.e., from observation 412 to observation 511). The ARIMA model yielded an RMSE of 1.053, while the base tree yielded an RMSE of 1.166. Thus, the ARIMA model significantly outperformed the base tree. (The last 100 values of the simulated time series ranged from -7.1940 to 4.4484, with a median of -0.8734 and a mean of -0.9318.)

Although the base tree demonstrates an ability to fit existing time-series data with reasonable accuracy, it fails to forecast well. Therefore, this result offers a baseline: any theoretically sound adjustments to the tree which yield an RMSE lower than 1.166 could indicate an improvement in the model; any adjustments which bring the RMSE below 1.053 would suggest that the model has the potential to outperform an ARIMA model.

When used to forecast, the regression tree performs two tasks: first, it fits itself to a training set, performing a number of splits, each designed to satisfy some objective function, subject to some penalty or minimum node size. Second, once the tree is fit, it predicts based on some additional data. The prediction for the base tree is simply the mean value of the leaf which the next-period observation wouldhave occupied.

In the base random forest model, the objective function is to minimize the sum of squared errors in a given node, as explained above (Section 1). Thus, in the base, vanilla tree, the nodes are designed to contain observations with similar values of . When the process which generates the data is unknown, this is a fine way to group the data. The purpose of the tree is to categorize a set of dissimilar data into more-similar subsets, and getting similar values of is one straightforward way to categorize. But if there is good evidence for a particular process which generates the data, then this approach doesn’t seem to work well. Indeed, if I know what process generates the data in general (e.g., an AR(1)), then what I want are not subsets of data in which the term has a similar mean, but subsets of data which are generated by similar particular implementations of the general process (e.g., all data is generated by some sort of AR(1), but one node contains observations where the coefficient is 0.9, and another contains observations where the coefficient is 0.75, etc.). So, if I assume the process is an AR process, then the objective function can become to minimize the sum of squared residuals on an AR(1) regression, rather than to minimize the sum of squared errors based on a mean expectation. In the base tree, I was agnostic about the process, so the most obvious way to fit the data was by using the mean value. Now, I have some intuition that the process is similar to an AR(1), so the most obvious way to categorize the data is by using an AR(1) function.

Now, the fitted values for each point in the dataset will be based on an AR(1) equation, fitted separately at each leaf. Each leaf will have a constant value and a coefficient value for the AR(1), and the fitted value of each observation in that leaf will be calculated accordingly. This is a much more flexible approach; the base tree had only as many unique fitted values as it had leaves, but this tree has a unique fitted value for each point in its training set.

Likewise, when the tree is used to predict the next-period value, that prediction will no longer be equal to the mean value of the leaf which that observation would have occupied. Instead, it will be based on what the fitted value would have been, given the constant and coefficient values associated with the leaf it would have occupied.

Thus, both the objective function used to fit the tree and the process used to forecast from the tree can be modified to accommodate the intuition that inflation data is similar to an AR(1) process. The table below shows the RMSEs that result from using four different models to generate fitted values and forecast values on the simulated time series. The first such model is the ARIMA optimized by AIC; the second is the base tree; the third is a tree which uses the same objective function as the base tree, but which offers unique predictions by assuming that the data in each leaf is generated by a similar AR(1) process; and the fourth is the tree described above, which uses AR(1) as an objective function as well to predict.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TABLE 1 | | | | | | | | | |
| Fit and forecast results on simulated AR(1) time series | | | | | | | | | |
|  | **Model Type** | | | | | | | |  |
|  | ARIMA | | Base Tree | | AR(1) prediction | | AR(1) objective function | |  |
| Fit RMSE | | 1.01629 | | 1.029163 | | 0.9716035 | | 1.015209 |  |
| Forecast RMSE | | 1.052803 | | 1.16591 | | 1.0922 | | 1.036251 |  |
|  | | | | | | | | | |
| Notes: The first column refers to the ARIMA function optimized by AIC; the last three columns refer to different types of trees: the base tree defined in Section 1, the tree which borrows the objective function from the base tree but predicts based on an AR(1) assumption, and the tree which uses an AR(1) objective function as well as an AR(1) prediction. The first row refers to the in-sample fit provided by each model; the last row refers to an out-of-sample one-period ahead forecast on the last 100 observations generated, in a time series of 511 observations. | | | | | | | | | |

The model which provides the best fit, by a fair margin, is the hybrid tree. When performing splits, this tree uses the mean-based objective function from the base tree as described in Section 1, but it uses an AR(1) expression for fitting and forecasting.

However, in the pseudo out-of-sample forecast on the last 100 values of the simulated time series, the best model is the improved tree described above. This improved tree even outperforms the ARIMA model, which is a promising result for the random forest I intend to construct. If the improved regression tree alone can outperform an ARIMA model even on a simulated AR(1) series, it’s reasonable to suppose that a random forest built from many of those same trees will be able to outperform an ARIMA model on real-world data.

* 1. *The Objective Function*

In the subsection above, I began to explore how well different types of tree models would predict on an AR(1) series. In doing so, I glossed over the specifics of how those tree models were implemented. The major difference between the base tree model described in Section 1 and the tree model which forecasts well above is the objective function. In Table 1, all of the tree models are self-tuning in terms of their penalty parameter; I describe and justify this process at greater length below. In this subsection, I describe the implementation and rationale for the AR(1) objective function.

*Implementing the objective function*. Within the initial node of the tree (i.e., the full dataset), the tree must make a decision which splits the data into two subsets. To optimize this choice, the tree refers to the AR(1) objective function. The function identifies one variable as the dependent variable (i.e., current period inflation ) and one variable as the first lag of that dependent variable (i.e., ). Then, for each of the remaining variables, the function seeks to perform the following optimization function.

Or, expressed in terms of an ordinary least squares regression,

In practice, the objective function takes all available observations and considers only three variables: the one remaining variable selected (e.g., ), the dependent variable, and the first lag. It sorts these observations in ascending order by the remaining variable selected (e.g., ). Then, at each observation, it splits the data into two subsets: one which contains all observations with below the split point, and the other which contains all observations at the split point and above. Within each of these subsets, it performs a least squares regression of on . It then calculates the sum of squared residuals for the regression at each subset and adds those sums together to get the sum of squared residuals which would result from that split.

The function performs this process for each unique value of the variable selected (e.g., ), so that every unique value of that variable is associated with a sum of squared residuals value. The function then outputs the minimum sum of squared residuals value and the value of the selected variable that corresponds to the split which produces that minimum. The function does this for each independent variable in the dataset (i.e., for through and the trend term), so that each variable is associated with a sum of squared residuals value that would result from the optimal split at that variable. The tree function then selects the variable and the split that generate the lowest sum of squared residuals and performs the split at that point.

The theory is that the two resulting nodes are now each characterized by the AR(1) regression that best fits them; the goal is to reduce the imprecise fit of the standard ARIMA function by splitting the data into subsets that are characterized by distinct AR(1) processes, thus improving the fit within each node and therefore the fit of the tree overall. In theory, this may provide too tight a fit and result in poor predictions, but that risk is mitigated by two concerns: first, if the entire dataset truly is generated by the same AR(1) process, then the penalty term within the tree will prevent any splits from occurring, and the tree will simply fit an AR(1) to the entire dataset. And second, I demonstrated above that, even in the case of simulated data which truly is generated by an identical AR(1) process from beginning to end, the tree can offer a better forecast than even an optimized ARIMA model. Thus, I am not concerned about overfitting the tree to the dataset by making “too many” splits.

*Optimizing the objective function.* The objective function could run up to one million times during the fitting process for each random forest. Therefore, it is worthwhile to code the function as efficiently as possible; even a small improvement in efficiency could result in hours saved, over the course of writing a paper such as this one. The practical details of how I optimized the AR(1) objective function can be found in Appendix 1.

* 1. *Tuning*

*Tuning in theory.* Machine learning in general relies on the idea that models can be self-tuning. Rather than require a human operator to specify parameters and values, the model trains itself. This includes assigning itself parameter values, which are usually optimized by a cross-validation technique. In the context of this paper, however, two special difficulties may arise. First, I have mentioned that any tuning technique will require some form of cross-validation; with cross-sectional data, it is straightforward to use a k-fold cross-validation technique. With time series data, however, the process requires more careful consideration. Second, the tuned trees will only be used to forecast in the context of a random forest that is itself tuned. Therefore, a perfectly tuned standalone tree may not actually be the best tree to use in the context of the forest. In other words, the tree cannot be tuned external to the forest, but must be self-tuning within the forest.

These are broad considerations about how to tune in general; there are also more specific concerns. For a random forest in the setting of this paper, there are impossibly many parameters to consider. For example, when sampling the data for each tree in the forest, should that sample occur with replacement? Without replacement? Starting from a fixed point in time? From a random point in time? How big would this random sample without replacement be? Should the sample be random at all, or should it actually be nonrandom? And none of these parameters can be tuned in a vacuum; each of these questions would have to be answered simultaneously with each other and with dozens of other such questions, regarding for instance the number of trees in the forest, the feature fraction for the forest, the splitting penalty for each tree, etc. Rather than get lost in the weeds of self-tuning so many parameters, I will focus in this section on merely one parameter: the penalty term for the tree.

The reason I focus on the penalty term for the tree is due to the second concern mentioned above: each tree must be tuned within the context of the forest. Some parameters, such as the feature fraction and the process for sampling data, I can and will specify, external to and independent from the particular context of each tree. For the penalty term parameter, however, the optimal value will vary depending on the data sample and on the features selected. Proper specification of the penalty value will be fundamentally important to the accuracy of the tree’s fit and forecasts, but that specification must depend on the inputs which the tree receives; the penalty value cannot be specified externally.

Now, in consideration of the second difficulty mentioned above, I have identified which parameters ought to be self-tuning. However, the first difficulty—that of actually designing the self-tuning mechanism—remains.

*Cross-validation.* In a typical machine learning model, a k-fold cross-validation technique would be used. The model would segregate data into *k* complementary subsets. Then it would designate one of these subsets as the test set, while the remaining (*k –* 1) subsets would together make up the training set. The model would be trained on the training set and then its parameters would be tuned based on its performance on the test set. This process would be repeated *k* times, each time holding out a different subset as the test set. This creates a pseudo out-of-sample forecasting test for the model, and allows it to self-tune. Instead of holding a fixed training set and a fixed test set, the model can optimize its performance across, say, ten different training/test set configurations. This makes the tuning more reliable than it would be in the case of just one training/test set configuration.

However, the random forest I construct is not a typical machine learning model; it is designed to forecast time-series rather than cross-sectional data. Time-series data may be serially correlated; the k-fold technique, which randomly sorts the dataset into different subsets, will therefore not be effective. Not only is time-series data serially correlated, but real-world time-series data is not generated by the same process across all points in time. In the case of US inflation data, for instance, Stock and Watson (2007) argue that the pre-1984 data is fundamentally different from the post-1984 data. On this evidence at least, if not for purely theoretical reasons, it would be best to avoid validating past predictions on future data.

Therefore, I use a cross-validation technique that accounts for the fact that time-series data is serially correlated. Each tree is given a certain set of data. Of this data, I withhold the most recent 11 observations as a test set for the tree, and I train the tree on the remaining, previous observations. Then, I test the tree on the 11 withheld observations in order to “cross-validate” it. For this model, the only parameter which self-tunes is the penalty term, so this is the parameter whose value is optimized by this validation process.

Each tree will be tuned independently after it receives a random subsample of both the data and the features. Thus, each tree in the forest will have its own fractional penalty term, which will be generated according to the process described in the previous paragraph.

*Tuning in practice.* I’ve explained the theoretical justification for tuning the penalty term, and for doing so using the specific validation technique described. Now I describe the implementation of this tuning.

I first specify a parameter space: the range of possible values for the penalty term. These values range from 0.70 to 0.99, increasing by increments of 0.005. In practice, the dataset is small enough that these increments are smaller than necessary: the difference between a penalty term of 0.905 and 0.910 is too subtle to make a difference in a dataset with only 733 observations. Indeed, these two penalty terms will almost always result in an identical tree. But, as previously discussed, in the case of this random forest it is better to over-specify than to underspecify—the model itself will be able to sort out which information is significant and which is not.

Now that the parameter space exists, the tree must be able to self-tune from the given options. It will do so according to the validation technique described above. The optimal penalty value is the one that minimizes the RMSE on the forecast for the 11 observations that make up the test set. Now the parameter space and the optimization function are both set; all that remains is to specify the search method.

The search method dictates how the model will search the parameter space. The most straightforward option is a grid-based search, where the model validates on each parameter value and selects the one that is optimal. However, this method is costly and inefficient. The parameter space I’ve specified for the penalty term contains 59 values. Thus, the tree model would have to fit 59 different trees on the training set and validate each of those 59 trees on the test set of eleven observations. Then, the model would identify the optimal penalty value and generate a tree on the complete dataset. Thus, a grid-based search method needs to fit 60 trees for each tree in the forest; in a forest of 50 trees, that means the model needs to generate 3,000 trees. This is costly.

Furthermore, it is inefficient. It is extremely unlikely that each penalty value results in a different RMSE, and that these RMSEs are distributed randomly across penalty values. Suppose the ideal parameter value is 0.850. It is far more likely that penalty values further from the optimal penalty value (e.g., values from 0.70 to 0.80, or from 0.90 to 0.99) yield relatively high RMSEs, while penalty values closer to the optimal value (e.g., from 0.825 to 0.875) yield relatively lower RMSEs. Thus, it is inefficient to dedicate as much time to searching the space from 0.70 to 0.75 as to searching the space from 0.825 to 0.875.

It would be better to identify promising regions of the parameter space and to explore those more thoroughly, while spending less attention on less-promising regions. An initial evenly-spaced search could provide a broad topography of the parameter space, then the next search could be more closely focused on the promising regions; the expected topography would then update, and a yet-more-closely focused search could take place, and so on.

This process is a Bayesian one, and I implement a Tree-structured Parzen Estimator (TPE) as described by Bergstra et al. (2011). For details on this process, see Appendix 2. The result is a much faster and a much more efficient search method, which the majority of the time selects penalty values identical to the ones chosen by a grid-based method, and which in my experiments always selects either the best or the second-best penalty value.

* 1. *Specifying the Random Forest*

The bulk of this section has dealt with the particulars of constructing the tree; now I describe the construction of the forest. I have exogenously specified many of the parameters in the forest: this forest consists of 50 trees, has a feature fraction of 0.7, and randomly samples for the length of the dataset which is fed to each tree. The penalty parameter within each tree is self-tuning, as described above, and is supplemented by the imposed stipulation that no node contain fewer than five observations.

The base tree from Section 1 had a 10-observation minimum, coupled with an imposed 0.9 penalty. However, due to the process of data sampling I am about to describe, I reduced that minimum to 5 observations, and allowed the penalty value to self-tune accordingly.

The “randomness” of the random forest consists in both the randomness of the data which is fed to each tree, and the randomness of the variables which are fed to each tree. In the case of the base forest, data is sampled with replacement from the full dataset. Given that a time series is serially correlated, this method of data sampling seems unlikely to yield the optimal result. Instead, using a technique called a “block bootstrap,” I randomly sample the *length* of the dataset, choosing a value between 50 and 100 from a uniform distribution. Each tree receives a dataset of a randomly selected length, whose last value is fixed at the most recent value in the series. For example, if I want to forecast inflation for January 2000, the “complete” dataset that the base forest would consider would contain all observations from January 1959 (the first month of data) to December 1999 (one month prior to the forecast month). The random forest I use, however, would take the last *x* observations from that dataset, where *x* is a random number from 50 to 100. Thus, only the most recent observations are fed to the trees in the forest. This maintains the spirit of the base random forest, namely that random sampling benefits the model, while preserving the integrity of the time series. The forest samples from the most recent observations because they are more likely to yield a tree that predicts well than observations from the 1950s are. Given the smaller sample size that this approach requires, I lowered the minimum node size from 10 in the base forest to 5 in this forest.

The data which each tree receives is randomly chosen, as I’ve just described, and so too are the variables which each tree considers. The variables in the base forest are through , plus a time trend. The feature fraction is designed to allow different variables to express their importance independently of each other, to mitigate the impact of variable correlation. For example, suppose that and are highly correlated (indeed, this is expected). In a model which always considers only when it also considers , the effect of the term is likely to be understated, since much of the impact is likely already accounted for in the term. Thus, the effect of may be overstated, while the effect of may be understated. The feature fraction mitigates this effect by allowing the model to sometimes consider both and together, to sometimes consider only and sometimes only , and sometimes to consider neither of them, as it considers other variables instead.

In this forest, the trend term and the term are always considered, in every tree. Of the remaining ten variables (i.e., through ), only seven are included in each tree. These seven are randomly selected. The trend is always included because it is unlikely to be correlated with any of the terms and may include information that the terms are unable to express. The term is always included because it is necessary in order to calculate the objective function and because I know that the data is at least partially generated by an AR(1) process.

Thus, each tree considers slightly different data and slightly different variables. Each tree will therefore generate slightly different fits and predictions for each observation. The forecast of the forest overall will be the mean average value of the predictions from each tree. The randomness of each tree mitigates the harms of misspecification and variable correlation. Taking the average of all that randomness results in a stable prediction that remains resistant to those harms.

1. RESULTS

I use the time-series-optimized random forest as described above to forecast US monthly inflation from January 1999 to January 2020. The optimized random forest outperforms the ARIMA model over that period. This result and its implications are discussed below.

*4.1 Goals*

The primary immediate goal of this project is to construct a random forest that outperforms an ARIMA model optimized by AIC. This is a simple but sophisticated model, and to outperform it would be no easy task. Secondary immediate goals are to outperform an AR(1) model, the “base” random forest described in Section 2, and a naïve one-month-ahead forecast. These goals are narrowly construed: the context is a one-month forward forecast of US monthly inflation data (as described in Section 1), from January 1999 to January 2020. Performance is gauged by RMSE.

An additional goal is to shed light on the use of random forest models in time series forecasting. This paper represents an exploration of a topic that is seemingly underexplored, and one which has great potential. In the next section, I will explore how effective the random forest model I’ve built is at predicting across different horizons and different datasets. For example, if the model I’ve constructed specifically to succeed at the task of predicting one-month-ahead US inflation from January 1999 to January 2020 is also successful at predicting, say, six-month-ahead UK unemployment from 1990 to 1999, that would indicate the broad potential of this approach.

*4.2 Possibility of Structural Break*

The time period I wish to forecast on is January 1999 to January 2020. This period is long enough that any success the random forest demonstrates is unlikely to be from random chance; and it is a period which includes the Great Recession, a challenging event to forecast around. I will be interested to see how well the random forest handles predicting inflation in the months before and after the initial plummet in inflation, and to see how it compares to the ARIMA model in the same months.

Despite the Great Recession, there is limited evidence of a structural break in the series. Indeed, in the larger series, which runs from January 1959 to January 2020, there is no evidence of a structural break at any point, according to a supF test (described by Hansen, 1992). Within the narrower space of January 1989 to January 2020, there is likewise no evidence of a structural break. (The random forest will only consider, at most, the 100 observations leading up to December 1998, and so won’t consider data before 1989.)

At 10-year intervals, beginning with the interval January 1989 – January 1999 and increasing by increments of one year, ending with January 2010 – January 2020, there is limited evidence of a few possible structural breaks. These structural breaks are of interest because they offer an explanation for the success of the model. If these structural breaks really exist, the model should be able to handle them better than the ARIMA does.

* 1. *Performance*

For the forecast period indicated, i.e., the one-month ahead forecast from January 1999 – January 2020, Table 2 shows the RMSE values for the time-series-optimized random forest model and four competing models. The table demonstrates that the random forest model described in Section 3 actually accomplishes the immediate goals outlined in Section 4.1 above; it outperforms every model listed in that section.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | TABLE 2 | | | | |  |
|  | RMSE Forecast results on US monthly inflation data, January 1999 – January 2020 | | | | |  |
|  | **Model Type** | | | | |  |
|  | Optimized Forest | ARIMA | AR(1) | “Base” Forest | Naïve |  |
|  | 0.002673159 | 0.002716793 | 0.002740606 | 0.002936135 | 0.003088432 |  |
| Notes: The numbers are RMSEs on a one-month forward forecast from January 1999 to January 2020, performed by five different models. The first column refers to the time-series-optimized random forest I describe in Section 3. The ARIMA model is optimized by AIC. The “base” random forest is as described in Section 2. The naïve forecast simply predicts that next month inflation will be the same as current month inflation. RMSEs for the random forest and the “base” random forest are not replicable. | | | | | | |

Due to the random nature of the forest, these results are not completely replicable. (In other words, the RMSE listed under the random forest entry above will not be exactly the same each time the forest is run.) However, the performance will tend to form a distribution of RMSEs that includes the RMSE listed above.

We see that the random forest outperforms the other models over the entire forecast period; Graph 1 shows how the models compare with each other throughout the period. In Section 3, I discussed at some length the value that the ARIMA model provides by offering a theory about the specific process which generates a time series. Now, I have discovered that the random forest forecasts better than even the ARIMA model. The graph below shows that the random forest offers good predictions when the ARIMA model offers good predictions, and poor predictions when the ARIMA offers poor predictions. There is no stark difference between the two, nor are there long periods where the one performs significantly better than the other.

With that said, the random forest’s pre-2010 performance, especially starting around 2007, helps it outperform the ARIMA model during the period overall. From 2009 onwards, all models begin to perform better, recovering from their poor performance of 2007 and 2008. The most significant takeaway from the graphs shown below is that the random forest and the ARIMA model, and indeed all five models, perform well at the same time and perform poorly at the same time as the others. No model performs significantly better during some periods and worse during others, relative to other models. This makes sense, since the random forest is built on an AR(1) objective function and uses an AR(1) prediction function; i.e., the forest is related to the AR(1) and ARIMA models. More surprising is that the “base” forest follows the same trend: it performs well and poorly when the other models perform well and poorly. This implies some sort of inherent “predictability” in the data; at times it is simply easier or harder to predict, regardless of the model.

|  |
| --- |
| Graph 1 |
| Cumulative and 12-month moving average of RMSEs from five models |
|  |
|  |
| Notes: These charts refer to the RMSEs produced by the inflation forecasts of five different models: the ARIMA model optimized by AIC, which appears in red; the time-series-optimized forest, which appears in black; the “base” forest, the simple AR(1) model, and a naïve one-month ahead shift, which each appear in a different shade of gray. Lower RMSE values indicate a better model. |

* 1. *Implications*

The model seems to be fairly successful at forecasting US monthly inflation at a one-month horizon from January 1999 to January 2020. This is great news! Just as the ARIMA model’s success (as opposed to the naïve model) justified a consideration of what made that model successful, so the success of this forest model as opposed to the ARIMA invites a discussion: what is going on in the model? What makes it so successful? Specifically, what makes it so successful compared to the base forest model described in Section 1?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TABLE 3 | | | | |
| Comparing base forest to time-series forest | | | | |
|  | **Model Type** | | **Improvement** |  |
|  | Base forest | Time-series forest |  |  |
| Forecast RMSE | 0.002936135 | 0.002673159 | -8.96% |  |
| Better fit than ARIMA: count | 114 | 128 | 14 |  |
| Better fit than ARIMA: frequency | 0.450593 | 0.505929 | 12.28% |  |
| Notes: This chart compares the “base” forest from Section 2 with the time-series-optimized forest from Section 3. In the second and third rows, both models are compared to an ARIMA baseline. The second row describes how many predictions, out of 253 in the forecast period, were more accurate than those of an ARIMA model. The third row describes this as a decimal value. | | | | |

Table 3 highlights the significant improvement made, both in terms of RMSE and in terms of frequency of good fits, as opposed to a baseline offered by the ARIMA model. Above, I discussed my expectations of how the model would perform and how the changes I made from the base forest would improve forecasting performance. Now, I dig into that improved performance and examine what accounts for it.

*Small tweaks to the AR(1) base.* First, as I noted above, the ARIMA model is very good. It performs well in forecasting situations, and it offers a plausible explanation for the process which generates inflation data. Thus, although I want to explore the benefits of the random forest model in the context of a specific time series, I need not start from a position of complete ignorance as to what generates my data.

Consider the number of “real” trees in each forest across the prediction period. By “real” trees, I mean trees which actually divide the data; i.e., trees which contain more than one node. In the base forest model, out of 50 typical trees, nearly all 50 of them will be real. By contrast, in my forest model, only about 4.6 trees per 50 trees are real. In other words, my model only actually decides to create a tree less than 10% of the time. In reality, of course, the model is performing tens of thousands of calculations behind the scenes in order to optimize the penalty parameter, but the penalty the model identifies as optimal, for a typical tree, is apparently less than 10% likely to yield any splits whatsoever.

Intuitively, this makes sense. If the data is actually generated by an AR(1) process or something similar, and if the default fit suggested by the tree is in fact an AR(1) process, then the tree will not split very often. If, on the other hand, the default fit suggested by the tree is simply the mean of all observations, then the tree is extremely likely to suggest splits when faced with a dataset generated by an AR(1) process. Indeed, this proves to be the case, and although the base forest fits a tree over 99% of the time (compared to 9.19% of the time for the time-series forest), the base forest nonetheless offers a reasonably good fit. I highlight this point to make it clear that, despite their reasonably similar fits (the forecast RMSEs are within 10% of each other), the two models are behaving in very different ways.

Specifically, the time-series-optimized forest isn’t doing much to change the AR(1) model which underlies it. For the more than 90% of trees which do not split, each tree assumes a single AR(1) process for the entire dataset; in other words, more than 90% of the time, the trees could be replaced by a simple AR(1) model. Recall from Table 2 above that the AR(1) model produces a worse forecast than the ARIMA model. Yet the time-series-optimized forest outperforms the ARIMA model. This performance boost is due to the 9.19% of trees which actually differ from a straight AR(1) model, and to the randomness of the features and data which the trees are fed. The forest doesn’t try to do too much; the AR(1) model is a good one, and the forest makes only occasional changes to it. By judiciously choosing when to alter the AR(1) model and when to let it alone, the forest outperforms, not just the AR(1) model itself, but also the ARIMA model. Small adjustments to a good model are all that is needed to improve it.

*Feature selection.* It is clear that the time-series-optimized random forest behaves differently from the “base” forest, and a look at the relative importance of different features in the two models drives this point home. A brief review: at each splitting point, a tree selects only one value of one feature to split by. This {feature, value} pair optimizes the objective function better than any other available {feature, value} combination does. Thus, it is reasonable to assume that the feature in question is somehow more influential than the other features when it comes to motivating the data at the particular node. In a regression, the size of the coefficient would indicate the “importance” of each variable, in terms of its influence on the data. In the random forest, the frequency with which each feature is selected may offer a similar assessment of feature importance.

“Feature importance” in this paper will refer to the frequency with which each feature is referred to within the leaves of a given tree (or a given forest). Based on this definition, the feature which performs the initial split in a tree will be given higher importance than a feature which performs a split later on. This is because each leaf refers to every split which led to the creation of that leaf, from the first split to the ultimate split. Thus, each leaf will refer to the feature which motivated the original split. Likewise, each leaf in the first branch of the tree will refer to the feature which motivated the original split as well as the feature which motivated the first split within the first branch, and so on. In this paper, “absolute importance” will refer to the number of times a feature is mentioned within the leaves of a tree. “Relative importance” will refer to the number of times a feature is mentioned within the leaves of a tree, relative to the number of times any feature is mentioned within the leaves of that tree.

I consider the “real” trees from every forest built for the forecast period January 1999 – January 2020, which amounts to 253 forests or 1265 total trees (not all of these trees will be “real” trees, however). Looking at only the “real” trees built during this period, I notice a pattern: the average “real” tree in the base forest assigns the highest importance to the feature , by a wide margin: this feature is mentioned 8.405 times per real tree. The time-series tree, by contrast, selects this feature only 1.178 times per real tree, or 14% as often. Meanwhile, the feature with the highest importance in the average “real” time-series tree was the *trend* term, which was selected 2.201 times per real tree.

In fact, in the real time-series trees, *trend* had a relative importance of 27.27%. Meanwhile, for the real base trees, *trend* had a relative importance of only 6.44%. The data paints a picture of a time-series tree which is both less deep and more intuitively focused than the base tree. That the base tree runs deeper than the time-series tree makes sense given that the base tree is working with a dataset that contains up to 729 observations, while the time-series tree is never working with more than 100 observations at a time.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| TABLE 4 | | | | | | |
| Comparing the average “real” base tree and the average “real” time-series tree | | | | | | |
|  | **Base** | | **Time-Series** | | |  |
|  | Absolute Importance | Relative Importance | Absolute Importance | Relative Importance | Relative importance vs. base tree\* |  |
| *trend* | 1.551 | 6.44% | 2.201 | 27.27% | 423.63% |  |
|  | 8.405 | 34.88% | 1.178 | 14.59% | 41.84% |  |
|  | 3.019 | 12.53% | 1.567 | 19.42% | 155.01% |  |
|  | 0.787 | 3.26% | 0.278 | 3.44% | 105.41% |  |
|  | 0.692 | 2.87% | 0.368 | 4.56% | 158.69% |  |
|  | 0.905 | 3.76% | 0.185 | 2.29% | 60.97% |  |
|  | 2.064 | 8.56% | 0.273 | 3.38% | 39.43% |  |
|  | 1.172 | 4.86% | 0.105 | 1.30% | 26.72% |  |
|  | 0.912 | 3.78% | 0.560 | 6.93% | 183.33% |  |
|  | 2.552 | 10.59% | 0.710 | 8.80% | 83.08% |  |
|  | 0.757 | 3.14% | 0.155 | 1.92% | 61.04% |  |
|  | 1.285 | 5.33% | 0.494 | 6.11% | 114.63% |  |
| **Total** | **24.102** | **(~100%)** | **8.073** | **(~100%)** | **---** |  |
| Notes: The absolute importance refers to the number of times a feature is referred to across all *leaves*. So, if a feature forms the basis for the first split, it will be referred to at least once in each leaf of that tree. Calculating importance in this way effectively weights for feature importance; a feature which forms the basis for the first split is more important to the model than a feature which is selected for a split at a depth of ten or twelve nodes.  \*The relative importance term of the average time-series tree as opposed to the relative importance listed for the average base tree. Normalizes for the fact that the base tree tends to be deeper. | | | | | | |

The table shows that, when normalized for the fact that the base tree tends to be deeper than the time-series tree, the time-series tree selects the *trend* term at over 4x the rate at which the base tree selects the *trend* term. On the flip side, the base tree selects the  term nearly 2.5x as often as the time-series tree does. The only other major discrepancy is with the term, which accounts for more than 12% of the features selected in each tree, but which the time-series tree selects at a rate over 1.5x as high as the base tree does. The other discrepancies in the table are likely coincidental, given that none of them involve features which have a relative importance of more than 10% of selections, from either tree.

All of this indicates that the base tree favors the term (which accounts for more than one-third of total selections in that tree), while the time-series tree favors the *trend* term muchmore strongly than the base tree does. This first fact actually makes some sense: the time-series tree wants to minimize the sum of squared residuals within a node based on the assumption that the data follows an AR(1) process. So, the importance of the first lag is already baked in. The base tree, on the other hand, wants to minimize the sum of squared residuals within a node based on the assumption that the data is fit to a mean value within each node. In this case, it makes sense that the tree will grab at the first lag term as the most important feature.

The second fact, however, is a little stranger. Why would the time-series tree favor the *trend* term where the base tree effectively ignores it (e.g., in the base tree, the *trend* term accounts for fewer selections than the term does)? To complicate matters, it’s not as though the time-series tree ignores the term, which it selects third most often of any term. So, it’s not as if the *trend* term necessarily cannibalizes the term in the presence of the AR(1) objective function. What, then, is going on?

There are probably a few explanations. First, the time-series tree is much less likely to split; only 9.19% of time-series trees in the forecasting period from January 1999 – January 2020 actually perform even one split. This, combined with the fact that the dataset fed to the time-series trees is much smaller than that fed to the base tree and the fact that the time-series tree uses an AR(1) objective function, means that any coincidental patterns the tree observes are much more likely to be motivated by the trend term than they would be in (1) a tree which is more comfortable performing splits, (2) a tree which considers a larger dataset, or (3) a tree which hadn’t already accounted for a reasonable effect from the term. The convergence of these three trees in one would probably yield something similar to the base tree, which selected the *trend* term at less than a quarter of the rate that the time-series tree did.

One possible interpretation is that the *trend* term tends to take on outsize importance at small sample sizes when the term is accounted for. Over larger samples, the *trend* term would not justify a split. Another interpretation is less dismissive: perhaps the *trend* term actually is more important than some models would indicate. Maybe in the short run, an AR(1) model can be augmented by including a *trend* term that passes a certain significance threshold. The model I’ve used certainly seems to offer an improvement over the ARIMA model (and over a simple AR(1) model), and this is effectively what it does: it assumes an AR(1) but allows for other variables if they meet a certain threshold (determined by the penalty term).

*Feature importance over time.* Above, I used feature importance to highlight some differences between the base tree and the time-series-optimized tree. But feature importance can be an extremely valuable tool for analysis as well. The features within a model that are on average more important than others are easily identifiable, and observable trends in feature importance over time offer clues to the changing process which the model suggests. For instance, if the *trend* term is favored by the model, with a relative importance of, say, 50% for the first several years of the forecast period, but then falls to a relative importance of only 15% in the later years, that might suggest a change in the underlying structure of the time series. For instance, it would indicate that short-run structural changes are more significant than autoregressive trends for the first few years of the forecast period, but then in the later years, the short-run structure is similar enough across about 100 months (about 8 years) that the autoregressive trends begin to become more reliable on their face, without requiring a break based on the trend term.

This is only an example, but Graph 2 tells a comparable story about US inflation data during the forecast period I have been considering. The graph shows a 12-month moving average of feature importance, displaying the top three features at every month. The *trend* term at first has a relative importance hovering between 40 and 50%, while and round out the top three features. The term is more important than the term, but not by much. Then, at about 2007, the *trend* importance jumps up to hover around 55%, and the and terms fall out of importance, in favor of the term. The remaining term is either or , depending on the particular month, but both these terms are fairly unimportant, never reaching more than 5% relative importance. Note that this is the non-sampled forest; i.e., it takes the entire dataset as input.

This analysis actually does suggest some kind of structural change right around 2006 or 2007. The *trend* term had been waffling between 40% and 50%; now it sticks tight at 55%. The other features had been and ; now both of those fade to irrelevance in favor of , while the third-most important feature never achieves more than 5%. This is certainly worth investigating further. It seems to indicate that during the first few years, inflation is a somewhat autoregressive process, while during the last few years, information about which month an observation is from is likely to be more valuable than information about what the previous month’s inflation was. Note that this holds only so long as the model is good; if the predictions which the random forest produces perform the same or worse than those of a simple AR(1) model, then this analysis is moot and there is no reason to suppose that the complex interpretation which the model suggests is any more accurate than the simple AR(1) process which the AR(1) model suggests.

Feature importance analysis is very intuitive, both in its presentation and in its performance. When I say it is intuitive in its presentation, I mean that its interpretation is straightforward; a glance at Graph 2 is sufficient to convey the point. And by intuitive in its performance, I mean that this type of analysis makes good sense as a valid way to think about the output of the random forest. The random forest does perform splits according to certain features; the features it selects to split by are in fact the features which have the greatest influence on the composition of the data at the time of the split. Previously, I had mentioned that machine learning tends to be thought of as a “black box”: a machine learning model takes input and produces output, without making any claims about the economy or justifying itself theoretically. The type of analysis presented in Graph 2 represents a release from that restricting framework: the random forest can actually tell us something about the economy. Through this type of analysis, the random forest model moves beyond its “black box” designation and into the realm of economic models.

|  |
| --- |
| Graph 2 |
| Relative feature importance from January 2000 to January 2020, 12-month moving average |
|  |
| Notes: This chart relies on the time-series-optimized random forest, with no data sampling. It displays the three most important features for each month, with a measure of their relative importance. The data displayed is a 12-month moving average. |

1. FURTHER APPLICATIONS

The random forest laid out in Section 3 performs well under a very specific set of conditions: it outperforms the ARIMA model on predictions of US monthly inflation data at the one-month horizon, where performance is measured by RMSE. This was the stated goal of this project, and it has been a success. However, the model was explicitly designed to function well under the precise conditions just described, so its success should not be a surprise.

If, on the other hand, the model designed to excel under the specific conditions outlined were to succeed under other conditions as well—say, in predicting a different time series, or predicting US monthly inflation across different horizons—then that would speak well, not just of my ability to compose a narrowly specified model which succeeds under particular circumstances, but of the ability of the random forest method to predict time series in general.

*5.1 The random forest in different circumstances*

*Different horizons.* The random forest put forward in Section 3 has proven to offer good predictions on US monthly inflation at the one-month horizon. In this section, I consider the predictions it offers at the 3-, 6-, and 12-month horizons. The results are summarized in Table 5. The time-series-optimized random forest compares favorably with the ARIMA optimized by AIC at all horizons, when it comes to predicting US monthly inflation data from January 1999 to January 2020. This result is encouraging, but expected, given that the model was designed with US monthly inflation in mind. The change in horizon does not affect the underlying process, nor does it affect the relative success of the forest model as compared with the ARIMA model.

*Different time series.* A bigger test will be how the model performs with different types of time series. Instead of US monthly inflation, I now consider different time series, at the one-month horizon. I consider three different time series, at three different periods: US monthly unemployment rate from January 1990 to January 2000, the US 3-month Treasury rate from January 1985 to January 1995, and the UK inflation rate from January 2015 to January 2020. The results are summarized in Table 5.

In the case of different time series, the random forest model compares perfectly well with the ARIMA or other models. This is an encouraging and unexpected result. I should note that neither the US unemployment rate series nor the US 3-month Treasury rate series are stationary. This presents a problem for every time-series model, not just for the time-series-optimized random forest. To make the data more workable, I de-trended and then differenced both series. The resulting series were stationary. I then predicted on that stationary series, using each of the four models in Table 5. Then I transformed the forecasts back to the form of the original series, by first un-differencing and next re-trending the forecasts. The RMSE values in Table 5 refer to the RMSEs from the resulting forecast. For the naïve values, I did not transform the data, but simply moved the existing time series one period forward. The UK monthly inflation data was stationary, so I did not transform it.

These results are encouraging, because they reflect good performance across different types of time-series data. The US monthly unemployment rate and the US 3-month Treasury rate are non-stationary and rather sticky, unlike inflation. The unemployment rate is not going to change very much from one month to the next, nor is the Treasury rate going to move as quickly and seemingly randomly as inflation is. UK inflation is similar to US inflation, but is still a fundamentally different time series, which may rely on a different motivating process. Despite these differences, the random forest outperforms a simple AR(1) model in all three cases.

I note the performance relative to the AR(1) model because the random forest I use is built on an AR(1) foundation. These results show that the changes I’ve made add value to the AR(1) model by judiciously choosing when to split the data into subsets and when to model the entire dataset as a single AR(1) process.

*Multivariate model.* The “univariate” model from Section 3 compares favorably with the univariate ARIMA model. Here, I consider how well a multivariate forest model of the same description as the one in Section 3 compares to a multivariate VAR model, when it comes to predicting US monthly inflation at a one-month horizon. The variables considered are a *trend* term and six lags on each of (i.e., through ), the 10-year US Treasury rate, the 3-month US Treasury rate, the unemployment rate, and the natural rate of unemployment, as given by the St. Louis Fed. The VAR referenced in Table 5 considers the same variables, each with six lags; so too does the “base” forest. The naïve forecast is simply a one-month ahead shift of the time series, as before.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | TABLE 5 | | | | | | | | | | | |  |
|  | RMSE Forecast results on different types of data | | | | | | | | | | | |  |
|  |  |  | **Model Type** | | | | | | | | | |  |
|  | **US monthly inflation data: horizons** |  | Optimized Forest | ARIMA | | AR(1) | | | “Base” Forest | | | Naïve |  |
|  | *1 month* |  | **0.002673159** | 0.00271679 | | 0.00274061 | | | 0.00293614 | | | 0.00308843 |  |
|  | *3 months* |  | 0.002990335 | 0.00308193 | | 0.0031898 | | | **0.00296987** | | | 0.00415206 |  |
|  | *6 months* |  | **0.002888543** | 0.0030296 | | 0.00317551 | | | 0.00289626 | | | 0.0041340 |  |
|  | *12 months* |  | 0.003019018 | 0.00303320 | | 0.00321220 | | | **0.00293278** | | | 0.00428745 |  |
|  | **One-month horizon: other time series** |  |  |  | |  | | |  | | |  |  |
|  | *US unemployment, Jan 1990 – Jan 2000* |  | 0.1412112 | **0.134985** | | 0.1426769 | | | 0.1404471 | | | 0.1402477 |  |
|  | *US 3-month Treasury rate, Jan 1985 – Jan 1995* |  | **0.2084125** | 0.2226686 | | 0.2091016 | | | 0.2344242 | | | 0.2398949 |  |
|  | *UK inflation rate, Jan 2015 – Jan 2020* |  | 0.00248008 | **0.00130273** | | 0.00287111 | | | 0.00249087 | | | 0.00401538 |  |
|  |  |  |  |  |  | | |  | | |  | |  |
|  | **US monthly inflation: multivariate model** |  | Optimized Forest | VAR | | | “Base” forest | | | Naïve forecast | | |  |
|  | *Predicted from Jan 1999 – Jan 2020* |  | 0.002690467 | **0.002097888** | | | 0.002863259 | | | 0.003088432 | | |  |
| Notes: The lowest RMSE in each row is bolded. For US monthly inflation data at horizons other than one month, the AR(1) model may be thought of as relying only on the most recent available observation of inflation in order to predict 3, 6, or 12 months in advance. The multivariate case substitutes a VAR of up to six lags, optimized by AIC, for the ARIMA optimized by AIC and excludes the AR(1) model entirely. | | | | | | | | | | | | | | |

* 1. *Implications*

The good news is that the random forest outperforms ARIMA models in forecasting US inflation, at every horizon. Furthermore, the random forest almost always outperforms a naïve model, in all contexts, and shows the ability to outperform the ARIMA model in the context of the 3-month Treasury rate. The bad news is that it does not outperform the ARIMA in every case, and occasionally does worse than the “base” forest, especially at different horizons. Worse yet, as a multivariate model it fails horribly, even offering a worse forecast than it did as a univariate model, not to mention being embarrassed by the VAR model.

But, this is a model specifically optimized for a particular forecasting scenario: US monthly inflation at the one-month horizon. Its success in other conditions is encouraging, and its failure cannot be too disheartening. With some modifications, it is likely to be able to challenge univariate models in any context. Some immediate ideas for improvement: eliminate or modify the data sampling procedure; maybe with different time series, different sampling techniques should be used. Change the feature fraction; possibly different time series call for a different degree of randomness in the model. Change the objective function; especially at different horizons, a different objective function may produce better results.

Whether the model can ever compete with multivariate models is an open question. Probably, it will tend to perform better in contexts with more data, and records of economic time series are a fairly recent innovation; i.e., they don’t contain many data points yet.

Interestingly, above, it seems like the base forest does not grow worse with a lengthening horizon, while the optimized forest and the other models do. I’m not sure why this is, but it once again suggests the forecasting potential latent in the random forest.

1. CONCLUSION

The random forest approach to time-series forecasting is promising. When specifically constructed to perform well in a certain context, it is able to outperform an ARIMA model. Even such a specifically constructed random forest is able to perform fairly well in other time-series applications; this suggests its potential for time-series use in general. The “base” random forest, too, performs quite well at times, outperforming an ARIMA model on US monthly inflation data at the 3-, 6-, and 12-month horizons.

The random forest in a time-series context is a topic which has been underexplored, but which promises to yield good results. Its potential is relatively untapped, and it has demonstrated the capacity to outperform classical univariate models. With some more adjustments, it is perfectly plausible that it could outperform multivariate models as well, and open the door to a new world of time-series forecasting.

Its value lies not just in its predictive power, but also in its ability to offer economic intuition through its expression of the relative importance of various features at various points in time. Graphs of feature importance such as Graph 2 above can indicate the presence and fluctuation of economic trends. Thus, the random forest should be valued not just for its predictive power, but for its ability to provide economic insight. I hope this approach is explored more thoroughly by other authors; its potential is vast.

**Bibliography**

Board of Governors of the Federal Reserve System (US), 1-Year Treasury Bill: Secondary Market Rate [TB1YR], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/TB1YR, September 10, 2020.

Board of Governors of the Federal Reserve System (US), 10-Year Treasury Constant Maturity Rate [DGS10], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/DGS10, September 10, 2020.

Board of Governors of the Federal Reserve System (US), 3-Month Treasury Bill: Secondary Market Rate [TB3MS], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/TB3MS, September 10, 2020.

“Coding Random Forests in 100 Lines of Code.” Statworx. June 5, 2019. Accessed October 9, 2020. https://www.statworx.com/blog/coding-random-forests-in-100-lines-of-code/.

“Coding Regression Trees in 150 Lines of Code.” Statworx. November 9, 2018. Accessed October 9, 2020. https://www.statworx.com/blog/coding-regression-trees-in-150-lines-of-code.

Federal Reserve Bank of St. Louis, 10-Year Breakeven Inflation Rate [T10YIE], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/T10YIE, September 10, 2020.

“Package ‘randomForest’.” CRAN. March 25, 2018. Accessed October 9, 2020. https://cran.r-project.org/web/packages/randomForest/randomForest.pdf.

Picard, Richard R., and R. Dennis Cook. “Cross-Validation of Regression Models.” *Journal of the American Statistical Association*, vol. 79, no. 387, 1984, pp. 575–583. *JSTOR*, www.jstor.org/stable/2288403. Accessed 8 Mar. 2021.

Tin Kam Ho, "Random decision forests," Proceedings of 3rd International Conference on Document Analysis and Recognition, Montreal, Quebec, Canada, 1995, pp. 278-282 vol.1, doi: 10.1109/ICDAR.1995.598994.

U.S. Bureau of Labor Statistics, Consumer Price Index for All Urban Consumers: All Items in U.S. City Average [CPIAUCNS], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/CPIAUCNS, September 11, 2020.

U.S. Bureau of Labor Statistics, Consumer Price Index for All Urban Consumers: All Items in U.S. City Average [CPIAUCSL], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/CPIAUCSL, September 11, 2020.

U.S. Bureau of Labor Statistics, Unemployment Rate [UNRATE], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/UNRATE, September 10, 2020.

U.S. Bureau of Labor Statistics, Unemployment Rate [UNRATENSA], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/UNRATENSA, September 10, 2020.

U.S. Congressional Budget Office, Natural Rate of Unemployment (Long-Term) [NROU], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/NROU, September 10, 2020.

U.S. Congressional Budget Office, Natural Rate of Unemployment (Short-Term) [NROUST], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/NROUST, September 10, 2020.

# “Using k-fold cross-validation for time-series model selection.” Stats Stack Exchange. August 10, 2011. Accessed October 9, 2020. https://stats.stackexchange.com/questions/14099/using-k-fold-cross-validation-for-time-series-model-selection

University of Michigan, University of Michigan: Inflation Expectation [MICH], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/MICH, September 10, 2020.

<https://www.r-bloggers.com/2019/09/time-series-forecasting-with-random-forest/> October 20, 2020.

<https://www.r-bloggers.com/2019/11/tuning-random-forest-on-time-series-data/> October 20, 2020

[CPIH INDEX 00: ALL ITEMS 2015=100 - Office for National Statistics (ons.gov.uk)](https://www.ons.gov.uk/economy/inflationandpriceindices/timeseries/l522/mm23)

February 20, 2021.

APPENDIX 1: OPTIMIZING THE AR(1) OBJECTIVE FUNCTION

R has a package to compute the sum of squared residuals (SSR) on any regression, but this is unnecessarily cumbersome for our purposes. Instead, the way the tree computes its optimal split point invites us to consider a more efficient shortcut. The tree operates by arranging the data in ascending order by some independent variable. Say for example that this variable is . Imagine the dataset sorted according to this variable. Now the data must be split at each value of that variable, in order to generate the SSR for each split. For the first configuration, imagine that the first subset contains 0 observations, and the second subset contains all the observations at the node in question (let’s call the number of total observations *n*). For the second configuration, the first subset will contain 1 observation, and the second subset will contain *n* – 1 observations, and so on. There will be *n* total configurations, with *n* total associated SSR values.

What is really notable about this setup is that each successive configuration differs from the last by *only one observation*. For example, the first subset in the first configuration differs from the first subset in the second configuration by only one observation: the first one. Likewise, the second subset in the first configuration differs from the second subset in the second configuration by the removal of only one observation: again, the first one. Each successive configuration is formed by moving one observation from the second subset of the previous configuration into the first subset of the current configuration; otherwise, the two subsets are unchanged across successive configurations.

This means that instead of recalculating the SSR at every data/observation pair, I can simply *update* the previous SSR. I retain the previous vectors and means, one each for above and below the previous splitting point. I identify the one observation pair that switches from the “above” vector to the “below” vector, and I update the previous vectors and means accordingly. Then I calculate the SSR, given the updated vectors and means. This method is much more efficient.

In practice, the relevant equation is this one:

Where is

When the formula is expressed in this way, it is easier to see how it may be updated. The SSR is, as the name implies, a sum. The only difference in that sum from one configuration to the next is the addition or removal of one and a corresponding . This in turn only affects the and terms, which may easily be updated. The computation of the SSR is now straightforward.

This update process is much more efficient than the process of computing, over and over again, the SSR for a series of independent regressions. Given that this objective function runs literally hundreds of thousands of times per forest, and therefore tens of millions of times across the forecast period, it is worthwhile to optimize it for computational efficiency.

APPENDIX 2: SEARCHING THE PARAMETER SPACE

*Overview*

To efficiently search the parameter space for the penalty term, I drew on the Tree-structured Parzen Estimator (TPE) technique described by Bergstra et al. in the 2011 paper “Algorithms for Hyper-Parameter Optimization.” For a full description of this process, see Section 4 of that paper, which lays out the necessary steps. For a less rigorous description, read on.

The quest to find optimal parameter values is referred to as “searching the parameter space;” in this case, the only parameter is the penalty term, so that space is one-dimensional. To search, the model runs with certain parameter values on a training set, scores its result against a test set, then runs with a different configuration of penalty values on the training set, scores that result against the test set, and so on. The goal is to find the configuration of penalty values that provide the optimal score.

This search may be performed in at least three ways: grid-wise, randomly, or in a Bayesian way. A grid-wise search would run the model or perform the desired task for *every possible combination* of parameter values (or, in the case of parameters which have continuous rather than discrete values, every possible combination of parameter values drawn at fixed intervals). In the case of my one-dimensional parameter space, the grid-wise search would generate one tree for each possible penalty value. In a one-dimensional space, the grid-wise search may not be very expensive, but with each added dimension, the search time has the potential to grow exponentially. On the plus side, this type of search basically guarantees that the model will identify the optimal combination of parameter values.

A random search would randomly sample parameter values and hope to cover the parameter space in that way. It need not be as exhaustive as the grid-wise approach, which is an advantage, but it may also fail to identify a set of parameter values which is anywhere near optimal.

The Bayesian approach is the most efficient; it may search randomly or grid-wise, at first, but eventually it updates the scope of the parameter space which it finds relevant to search; in other words, it focus narrows more and more closely as it continues to run, until it identifies a set of parameter values which is sufficiently optimal.

*The TPE Approach*

The TPE approach I use is a Bayesian one, and operates as follows. First, the tree performs a wide, grid-wise search of the parameter space. Possible penalty values range from 0.70 to 0.99, by intervals of 0.005. Thus, there are 31 penalty values. The tree’s first search uses the penalty values 0.70, 0.75, 0.80, 0.85, 0.90, and 0.95. It generates a score for each of these parameter values, based on how well they predict on the 11 observations which are withheld.

Once the model has six scores to correspond with the six different parameter values, it sorts the parameters into two groups: an *l* group and a *g* group. The *l* group consists of the 50% of parameter values which correspond with the best scores; the *g* group consists of the remaining 50%. In theory, then, after the first search, there are 3 parameter values in the *l* group and 3 parameter values in the *g* group. The *l* group is used to create a distribution, with the 3 values which make up the group forming 3 peaks in the distribution; each peak resembles that of a normal distribution, with a certain standard deviation. The same is done for the *g* group.

Then, ten penalty terms are randomly selected from the *l* group’s distribution. Each of the ten penalty terms is evaluated on the following basis. It is checked against the distribution of the *l* group, and its value in that distribution is found. This may be called its *l* value. Then it is checked against the distribution of the *g* group, and its value in that distribution is found; this is its *g* value. Its *l* value is divided by its *g* value, and the resulting value becomes its score. Each of the ten penalty terms is scored.

The score is supposed to represent the model’s expectation for how well the model would perform with that penalty term. If the *l* distribution consists of the “good” penalty values, and the *g* distribution consists of the “bad” penalty values, then a value’s score on the *l* distribution answers the question, “If I were to choose a good parameter value, how likely would I be to choose this particular value?” Likewise, the value’s score on the *g* distribution answers the question, “If I were to choose a bad parameter value, how likely would I be to choose this particular value?” To divide the *l* score by the *g* score is to combine both answers into one; the higher the total score, the more likely the particular value is to be a good one.

So, ten values have been sampled from the *l* distribution, and they have been preliminarily evaluated as described. Of these ten, the three values with the highest score are actually tried out; the model runs three times, with these three parameter values, and the model performance is scored for each of these three parameters, based on how well they handle the 11 observations which are withheld.

Now the model has run 9 times: the first 6 times for the broad, grid-wise search, and 3 additional times as described above. The model now has experience of 9 parameter values, and a corresponding score for each value. Accordingly, the model now sorts those 9 values, just as it had sorted the 6 values before: the 50% of parameter values which correspond with the best scores go to the *l* group, and the rest go to the *g* group. Now the *l* group and the *g* group each have four or five parameters in them, with corresponding performance scores.

Next, the model draws ten parameter values from the *l* distribution, scores them against both distributions as before, and selects the three values to actually run, just as before. This process occurs 5 total times, so that the model runs 21 times: 6 times initially, and then 3 more times on 5 additional occasions. This is already 10 times less than the grid-wise approach, and in fact the benefit is even greater than that: the parameter values which the model decides to run will often be repeated, and in this case the model will simply duplicate the result from the last time it was run with that parameter value. So, in practice, the model will likely run much less than 21 times: perhaps as few as 12 or so.

Finally, the model has run with different parameter values and evaluated its performance on each occasion. Now it selects the parameter value which scored the best, and generates a tree from the complete set of data which it is fed: both the erstwhile training set and the erstwhile test set.

APPENDIX 3: CREATING THE DATASET IN R

To convert the time-series data into a form which is usable by the random forest model, I use the embed() function in R to convert the time series into a matrix of values. The first column is a vector of inflation observations, from January 1960 to September 2020. The second column is the same vector, from the prior month: December 1959 to August 2020. The third column is one month prior again, November 1959 to July 2020, and so on. The first column represents the values, the second column , the third *,* and so on, for each of more than 750 observations. Each lag on an observation then, is treated as a feature in the data which is used to train the random forest. I use 11 lags. I also append a time trend term, which contains a value of 1 for the first observation, 2 for the second, etc. Each row now consists of one observation, containing a term with 11 lags and a time trend.

Just as the random forest’s feature fraction is supposed to prevent overfitting the data, the regression tree’s splitting criterion also prevents overfitting. At each node, only one feature will ultimately determine the split. Thus, for a data set in the form of the matrix described above, it is theoretically impossible to overfit by adding too many features. Even if the matrix had 100 lags, the splits at each node would only be determined by the one feature whose choice minimized the sum of squared errors. It is very unlikely that this feature would be the 100th lag, at any node. Eleven lags in an ARIMA model on this data would likely lead to an overfit; the best ARIMA model by AIC is an ARIMA(4,0,1). But, since the regression tree self-identifies the most important features, and allows only them to determine its fit, there should be no harm to adding more lags than are likely to be necessary.

APPENDIX 4: JUSTIFICATION FOR FRACTIONAL PENALTY

Given the increased flexibility that the fractional penalty provides, it seemed like the best choice. In a regression tree, it is inevitable that some nodes will contain observations that are very similar, while others contain observations that are only somewhat similar. For example, at the very first split, when all the data is under consideration, it is possible that there really are two clearly distinct groups in the data. The first group represents about half of the data and consists of nearly identical observations. The second group, which likewise represents half of the data, is remarkable for the relative dissimilarity of its observations. Assume that the split is performed correctly, so that the two groups are situated in two different nodes. The first node may very well require no further splits. The second node, on the other hand, consists of observations which have not much in common. It was formed merely by default, the leftover result of the split which so appropriately segregated the first group. Here I have a case where some nodes contain observations which are very similar, while some nodes contain observations which are very different.

If a node contains observations which are very different, I would want those observations to be further split out, until I had nodes which contained relatively similar observations. In the case of a fixed penalty, this may not occur. A large node may contain relatively similar observations, but because of its sheer size, it reduces the SSE by a given amount when it is split. On the other hand, a smaller node further down on the tree may contain relatively dissimilar observations, but because of the node’s small size, a split at this node would only reduce the overall SSE by so much.

With a fractional penalty, this would not be the case. The large node of similar observations may possess an absolutely larger SSE within the node than the smaller node of dissimilar observations possesses. But the relative reduction in SSE when splitting the smaller node as opposed to the larger node would be greater; thus, the split at the smaller node would be valued more highly than the split at the larger node. This seems appropriate.

With this in mind, I opted for the fractional penalty. However, I did not know exactly what the fraction should be. Thus, rather than stipulate a specific fractional penalty, I opted to allow the tree to self-tune. Looking back, given the wide range of optimal fractional penalty values chosen by different trees in different forests, this decision seems wise.