ALEX MORAN

# Predicting U.S. Monthly Inflation with Random Forest

I explore the process of creating and tuning a random forest model designed to compete with classical time series models. The goal is to generate a “univariate” model that can outperform a standard ARIMA model 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: 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 AR and ARIMA models are useful and valuable, both for their forecasting ability and for their specification of a time series’ underlying process. But machine learning models such as the random forest also have the potential 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. It proposes a modification of the standard random forest proposed by Ho in 1995, in order to account for the peculiarities of time-series data as opposed to cross-sectional data. The modified random forest presents a significant improvement over the standard random forest, both in terms of fit as measured by root mean squared error (RMSE) and in terms of forecast value as measured by root mean squared forecast error (RMSFE). 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 modified random forest is that, even given its moderate success in fitting and forecasting the US monthly inflation time series, the random forest as a model does not yield theoretical insight about that time series. The random forest does not 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. Although the success of the modified random forest cannot be projected directly into a theory about the mechanisms motivating inflation in the US, I can and do consider the reasons for its success 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” in that it takes as input only lags of inflation, and 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, as a “univariate” model for monthly inflation across different prediction horizons, and even for other types of time series: foreign inflation data, and domestic unemployment rates, 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 random forest as a concept in more detail, Section 2 describes the data used for the bulk of the paper, 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 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.

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. That pattern is 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. Of course, 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 splitting point; 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”. Within each resulting node, the model starts the same process over, and so on, until a complete tree is formed. Generally, the depth of the tree is limited by some imposed parameter (e.g., each node must contain at least 10 observations).

Each node, including each terminal node (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

All observations in the training set which pass those filters are assigned to the same node. If the node is a terminal node, the subset of observations which it contains will not be split further. 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.

Within each terminal node, the tree will compute the mean of all values. This 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.

As mentioned above, the depth of each tree is generally limited by some 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 740 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 (designated as lambda). 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 bring the sum of squared errors within its own node to a value of at most 0.90 of the original sum of squared errors within that 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 lambda penalty term will result in a much shallower tree than a higher penalty term. The choice of lambda will depend on the data and the model.

By itself, the 0.9 penalty term would yield a fairly deep tree, 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. However, within the context of the forest, there is another parameter to consider: the feature fraction. This parameter refers to the fraction of features (variables) that are used in creating each tree. In the base model, the feature fraction is 0.7. 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; setting the feature fraction appropriately is essential to maintaining that strength. If it is set too high, the gains in accuracy may come at the cost of overtraining. If it is set too low, the model may fail to achieve its full possible accuracy.

In the base model, each tree in the forest is trained on a random sample of features, the size of which is determined by the feature fraction. It is also trained on a random sample of the available data, which is sampled with replacement from the original data and contains as many observations as the original data. Both these elements of randomness are supposed to prevent overfitting while preserving or improving accuracy.

The base model contains 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. 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 suffers from seasonality, trends, and covariance, 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 try to make the data as least tricky as possible. The data I use is seasonally adjusted. 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. There is no significant evidence of a structural break, assuming that the data follows a general AR(1) process. 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.

Once I’ve established this, I take two further steps. First, as a matter of practicality, I make the data compatible with the random forest function by using 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.

Secondly, after I’ve transformed the data into this matrix form, I retool the random forest I have described above to make it more effective with time-series data. The random forest method is not expressly designed to handle time-series data, but with a few adjustments, this method could present an improvement over a classical ARIMA model. To realize that potential improvement is the goal of this project, and the adjustments I employ to that end are outlined in the next section.

1. REIMAGINING THE RANDOM FOREST

I need to reimagine the random forest, since I am dealing now with time-series data and not with cross-sectional data.

*3.1 Considering the Underlying Process*

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

The ARIMA model does a good job of predicting future inflation. The ARIMA model is an autoregressive integrated moving average model. It 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 not just offer predictions about the future; it offers, in the abstract, a theory of what sort of process motivates the data (an autoregressive moving average process); and it offers, in its implementation, a specific hypothesis about exactly which autoregressive moving average process motivates 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 spits out 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 motivated 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 motivated by the process which the ARIMA model describes.

Armed with this information, I don’t have to remain agnostic about the process which motivates the data. Instead, as I construct the random forest model, I can assume that the data is partially motivated 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 motivated 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 motivated 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 411 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 split is 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.

The core function in the regression tree is referred to as the “objective” function. It defines some object to achieve by performing splits. 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 motivates the data is unknown, this is a fine way to group the data. The point of the tree is to categorize a big mess 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 motivates the data, then this approach doesn’t seem so good. Indeed, if I know what process motivates 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 motivated by similar particular implementations of the general process (e.g., all data is motivated 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 fit 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 motivated 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 |  |
|  | | | | | |

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 great news 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 operates as follows.

It 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 motivated 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 motivated 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.

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 presumes a serial correlation between successive observations; 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 the same 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 lambda, 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 lambda. 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 740 observations. Indeed, these two penalty terms will almost always result in an identical tree. But, as previously in this paper, 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 would need to fit 60 trees for each tree in the forest; in a forest of 50 trees, that would mean the model would need 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.

*3.5 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 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, 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 motivated 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.

Given the increased flexibility that the fractional penalty provided, this 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 accurately 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.

1. RESULTS
   1. Possibility of Structural Break
2. IMPLICATIONS
3. CONCLUSION

**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.

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

APPENDIX 1: OPTIMIZING THE AR(1) OBJECTIVE FUNCTION

*Optimizing the objective function.* Thus in a dataset with a dependent variable and ten independent variables and for a tree which ends up having 100 total nodes (both terminal and non-terminal), the objective function will be run 1,000 times. There are fifty trees in the forest, so even in the simplest case, the objective function will be run 50,000 times to generate a single forest. Ours is not the simplest case, however; each tree is self-tuning, which means that each tree is run and re-run dozens of times before it enters the forest. Assume this occurs twenty-five times, and the objective function will be run 1,250,000 times to generate a single forest.

Given limited computational power and time, optimizing this function is key. A difference in efficiency that is practically unnoticeable when the function is run 1,000 or 10,000 times will become unbelievably stark when the function is run 1,250,000 times. The obvious approach to choosing a split which minimizes SSE is to consider the vector of the dependent variable together with each of the vectors of dependent variables. For each vector of independent variables, the function would follow the same process: Sort the data pairs by increasing order in the independent variable. At each data pair, calculate the SSE of the dependent variable below that pair. Calculate the SSE above and including that pair. Add the SSEs together. Append that SSE to a vector; the vector will be equal in length to the two vectors considered. Identify the minimum SSE in that vector. Identify the value of the independent variable which corresponds to that SSE.

If I were to split according to that independent variable, the optimal split would occur at that observation. I calculate the optimal split and resulting SSE for each of the independent variables. Then I identify the minimum SSE from that collection, along with the corresponding variable and value. I perform the split. This method is rather clunky. Instead of recalculating the SSE at every data/observation pair, I can simply update the previous SSE. 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 SSE, given the updated vectors and means. This method is more than twice as efficient.

APPENDIX 2: BAYESIAN TPE OPTIMIZATION