Time Series Forecasts with Random Forest

Note to Dr. Kolda: This is the text draft of my thesis, but most of the work has been done in R. A link to that code and data is here: https://github.com/amoran57/thesis.

The questions:

- How to use random forest with time series?
    - Is it possible to overfit?
- How to tune the parameters?
    - Feature fraction
    - Minimum size vs. other criteria (for leafs)
- Applications
    - Significant variables (and how these change with horizon)

For tuning:

- Pick a few hyperparameters
- Select values for those hyperparameters
- Use cross-validation to choose best values

Abstract
This paper details the process of creating and tuning a random forest model designed to compete with classical time series models. The goal is to generate a model that can outperform a standard ARIMA model at a one-month forecast horizon. The paper deals with issues related to creating and tuning the model, identifies areas for improvement, and outlines advantages and disadvantages to using a random forest model for time series data. The purpose of the paper is to compare the forecasting potential of the machine learning model to that of some more-traditional models. In general, the random forest model does outperform the others. This is especially true at lower forecasting horizons.

I. Introduction

Predicting the future is hard. It makes sense, then, to explore all of the resources at our disposal when we set out to forecast. We should not limit ourselves to a certain class of models, as if only these could reasonably be expected to forecast well. Indeed, some classical models do a fair job, but no model is perfect, and many models even fare worse than a naïve forecast.

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, it makes sense to at least consider applying some machine learning techniques to time series forecasting. Classical 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.

II. 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. 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 risking overfitting—an ability which classical time series models lack.

The random forest we will use is more simplistic than Ho's. For each regression tree, the splitting criterion is as follows. For each feature (i.e., variable) in our data set, the model arranges the data in ascending order. Then, it identifies the observation at which a split will minimize the sum of squared errors for the data. It records this 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 model selects the one that minimizes the sum of squared errors and then performs the split. Within each resulting node, the model starts the same process over, and so on, until a complete tree is formed.

Each node, including each terminal node (leaf) is characterized by some splitting filter or series of splitting filters. An example of such a series may be

$$\pi_{t-1} >= \ 0.00723766730023012 \,\&\, \pi_{t-6} >= \ 0.00590843668616614 \,\&\, trend \ > \\ = \ 222 \,\&\, \pi_{t-1} \ < \ 0.0127390257774298$$

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. At the terminal node, the tree will compute the mean of all π_t values. This mean will be the fitted value for the training set, and the predicted value for any observation in the test set which happens to satisfy the filters above. Note that π_{t-1} 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.

For each tree, the predicted π_t value on a new observation will be equal to the mean of the π_t values of observations at the leaf which the new observation would occupy. In the random forest, the predicted π_t 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 new observation's π_t value. The forest's prediction will be the mean of those predictions.

Within each tree, the splitting criterion is optimized by some fractional penalty term lambda, which ranges from 0.75 to 0.99. The penalty requires that splits bring the sum of squared errors *within the current node* (not overall) to a value of at most 0.75 (for example) of the original sum of squared errors *within that node*. The choice of lambda will depend on the data and the model.

The penalty term lambda affects the depth of the tree. The higher lambda is, the less gain in accuracy is required to justify each split, meaning more splits will occur and the tree will be deeper. With a lower lambda, the tree may be quite shallow. Thus, a higher lambda will fit the tree more closely to the training set, risking overtraining. A lower lambda will have the opposite effect.

The other parameter is the feature fraction. This parameter is fixed for the entire forest, whereas the penalty term may vary with each tree. The feature fraction refers to the fraction of features (variables) that are used in creating each tree. In our model, the feature fraction ranges from 0.3 to 0.9. Each tree within the forest will be created using a random sample of 0.3 (for example) 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.

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 model we use will 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 our resources are limited. Fifty trees should be enough to guarantee a reasonable level of accuracy.

III. The Data

Our data is a time series: United States CPI monthly inflation, seasonally adjusted, 1959-2020. Time series data is notoriously tricky. It suffers from seasonality, trends, covariance, and a myriad of other maladies. Many of the classical models 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, we try to make the data as least tricky as possible. The data we use is seasonally-adjusted. This should account for seasonality issues. It is also stationary, according to a unit root test. <u>There is no significant evidence of a structural break or a time trend.</u>

To make the data compatible with the random forest, we 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 π_t values, the second column π_{t-1}, the third π_{t-2}, 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. We use 11 lags. We also append a time trend term, which contains a value of 1 for the first observation, 2 for the second, etc.

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. Thus, unlike the standard ARIMA model, the regression tree self-identifies the most important features, and allows only them to determine its fit.

The random forest method is not expressly designed to handle this sort of data. However, the random forest method can be very effective at predicting cross-sectional data, and it is not impossible that with a few adjustments and with some careful handling, this method could present an improvement over a classical ARIMA model. To realize that potential improvement is the goal of this project.

IV. Tuning

The bulk of this project consists in appropriately tuning the forest and the trees. We anticipate two special difficulties that may arise when using the random forest method on inflation data. First, an especially prominent challenge that results from the time-series nature of our data is the problem of cross-validation. 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, we realize that our tuned trees will only be used to forecast in the context of a random forest that we also tune. Therefore, a perfectly tuned standalone tree may not actually be the best tree to use in the context of the forest. How do we tune the trees and the forest together? This is something we need to consider.

In addition to these two concerns, we realize that there are impossibly many parameters to consider. For example, when we randomly sample the data for each tree in the forest, should we sample with replacement? Randomly sample without replacement from a fixed point in time? From a random point in time? How big would this random sample without replacement be? 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. Thus, to perfectly tune the forest would take an extraordinary amount of computation, which we do not have access to. So, we will focus our efforts on tuning the parameters outlined in the second section, above: the fractional penalty on splits and the feature fraction.

Instead of the k-fold cross-validation technique favored when training models on cross-sectional data, we will use an alternative that makes more sense for time series data. For all we know, the time series is fundamentally different in the 2010s as opposed to the 1960s. Maybe there's a structural break, maybe there's some kind of non-linear time trend; at any rate, it would be best to avoid validating past predictions on future data. Therefore, we will hold the most recent 48 observations as a test set for each tree and train each tree on the previous hundreds of observations. Then, we will test the tree on the last 48 observations in order to "cross-validate" it. Within the forest, each tree contains a random subsample of data and features. This means that the last 48 observations in the data may not (indeed, almost certainly will not) be 48 successive observations. Some will be repeated, some will be skipped. Nevertheless, the tree which results from the training set will at least be tested on later data, rather than earlier data.

The most prominent parameter in a regression tree is the splitting criterion. The most prominent parameter in the random forest is the feature fraction. A standalone tree could be very well-tuned to the data sample and the entirety of the features, but we want the tree within the context of the forest to be self-tuning to the data and features that it is given. Therefore, 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.

V. Working Through the Process

We began by sourcing some basic random forest R code from the Statworx website. Then we proceeded to customize it. The initial code had a function for the regression tree and a function for the random forest; the random forest function contained a function which was meant to "sprout" trees by calling the regression tree function.

The core function in the regression tree is referred to as the "objective" function. It defines some object that we wish to achieve by performing splits. In our case, the objective is to minimize the sum of squared errors in a given node, with the added condition that the split also satisfy our fractional penalty. But the key feature of this objective function is determining which split minimizes the sum of squared errors; the penalty may be added. The objective function must be called for every feature at every node in the tree.

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 our 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 we were to split according to that independent variable, our optimal split would occur at that observation. We calculate our optimal split and resulting SSE for each of the independent variables. Then we identify the minimum SSE from that collection, along with the corresponding variable and value. We perform the split. This method is rather clunky. Instead of recalculating the SSE at every data/observation pair, we can simply update the previous SSE. We retain the previous vectors and means, one each for above and below the previous splitting point. We identify the one observation pair that switches from the “above” vector to the “below” vector, and we update our previous vectors and means accordingly. Then we calculate the SSE, given the updated vectors and means. This method is more than twice as efficient.

Having optimized the objective function, we turn to customizing and tuning the tree. The initial regression tree function did not contain a penalty function to limit the number of splits; instead, it employed a minimum size requirement for each node. Within a given node, it would determine the optimal split to perform. Then it would check the size of the two nodes that would result from that split. If either of the resulting nodes contained less than, for example, 10 observations, the split would not occur. The original node would be marked as a terminal node, and the program would move on to consider the next node.

We had a choice for how to limit the number of splits performed. We could either keep the minimum node size stipulation, we could specify a maximum number of nodes or terminal nodes that each tree should contain, or we could implement a penalty on each split. The penalty is the most sophisticated of these methods, since in principle the same penalty allows for a very deep or a very shallow tree, depending on what best suits the data. The other options would tend to result in trees of consistent depth, regardless of the data.

But we were still faced with a choice: which type of penalty to use? We could set a fixed penalty, such that the SSE would need to be reduced by a certain, fixed amount in order to justify a split. Since we would allow this fixed number to be tuned for each tree, we could achieve a reasonable fit this way. Or, we could implement a fractional penalty, which is even more flexible. It would require that each split reduce the SSE by at least a certain fraction, probably in the 0.05-0.15 range. If the split failed to reduce the SSE by that fraction, the split would not occur and the node would be marked as a terminal node.

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 we 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, we would want those observations to be further split out, until we 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, we opted for the fractional penalty. However, we did not know exactly what the fraction should be. A core tenet (indeed, the core tenet) of machine learning is that models be capable of self-tuning. Thus, rather than stipulate a specific fractional penalty, we 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.

When tuning the forest, it's important to consider the trees and the forest itself as complementary entities. In other words, neither the tree nor the forest should be tuned in a vacuum. Instead, the tree should be tuned with the understanding that it will be used in a forest explicitly designed to accommodate overfitting. Thus, we may allow the tree to fit the data more tightly than we would if we planned to use the tree alone for forecasting.

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