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 753 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 our 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 CPI monthly inflation, seasonally adjusted, 1959-2020. Time series data is tricky: it suffers from seasonality, trends, 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. It is also stationary, according to a unit root test. There is no significant evidence of a structural break or a time trend.

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.

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.

Secondly, 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. 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, 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 are 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; indeed, with cross-sectional data, there may not be such a process. We, however, have an intuition that a time series like inflation is motivated by some sort of autoregressive process. How should that intuition 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 least squares regression on lags of itself; it is “integrated” in that it automatically differences the data 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 us *why* our data behaves the way it does. The predictions are so 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 simply ignore the good information the ARIMA model provides. The whole spirit of machine learning is to use every tool at our disposal to generate the best model possible. In this case, I want to generate the best random forest model possible. Now, for our data, I find that on a one-month forward prediction of US monthly inflation, from January 1999 to January 2020, the ARIMA model yields an RMSE of 0.0027, while a naïve model 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 start from scratch now when building the random forest model. Instead, 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 our 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 generate 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 Adjusting the Tree*

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. For each existing value of each independent variable, the objective function calculates the mean of dependent variable values below that and computes the sum of squared errors based on how each observation below the split differs from the mean of all values below the split. Then it does the same calculation for values above the split point. The total of the sum of squared errors below and above the split is the SSE associated with that split. The objective is to minimize that SSE value.

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 our data in general (e.g., an AR(1)), then what I want is 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.)

*3.3 Tuning*

The bulk of this project consists in appropriately tuning the forest and the trees. Two special difficulties may arise when using the random forest method on inflation data. First, an especially prominent challenge that results from the time-series nature of the 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, 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. How can the trees and the forest be tuned together? This is something to consider.

In addition to these two concerns, there are impossibly many parameters to consider. For example, when randomly 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? 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 computational power, which I do not have access to. So, I will focus on tuning the parameters outlined 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, I use an alternative that makes more sense for time series data. 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 hold the most recent 11 observations as a test set for each tree and train each tree on the previous hundreds of observations. Then, I test the tree on the last, withheld 11 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 (i.e., the penalty term or the minimum leaf size). 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 it is better for 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.

*3.4 Coding the Random Forest*

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 the objective function. 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.

1. RESULTS
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