Time Series Methods in the R package **mlr**

Steve Bronder

sab2287@columbia.edu

December 29, 2016

Abstract

The **mlr** package is a unified interface for machine learning tasks such as classification, regression, cluster analysis, and survival analysis. **mlr** handles the data pipeline of preprocessing, resampling, model selection, model tuning, ensembling, and prediction. This paper details new methods for developing time series models in **mlr**. This extension includes standard and novel tools such as Lambert $W \times F()$ transform data generating processes, autoregressive data generating schemes for forecasting with machine learning models, fixed and growing window cross-validation, and forecasting models in the context of univariate and multivariate time series. Examples are given to demonstrate the benefits of a unified framework for machine learning and time series.

1 Introduction

There has been a rapid development in time series methods over the last 25 years [23] whereby time series models have not only become more common, but more complex. The R language [39] has several task views that list the extensive amount of packages available for forecasting, time series methods, and empirical finance. However, the open source nature of R has left users without a standard syntactic framework whereby individual packages will have a sub-culture of style, syntax, and output. The **mlr** [7] package, short for Machine Learning in R, is a meta-package which works to give a strong syntatic framework for the modeling pipeline. By automating and standardizing the tools and methodologies in machine learning, **mlr** reduces error from the user during the modeling process.

This extension to **mlr** is the first R package, to this authors knowledge, which allows for a fully standardized framework for rigorously testing and training of forecasting models. While there are some time series methods available in **caret** [18], development of forecasting models in **caret** is difficult due to computational constraints and design choices within the package. The highly modular structure of **mlr** makes it the best option for implementing time series methods and models. This paper will show how using **mlr**'s strong syntactic structure allows for time series packages such as **forecast** [28], **rugarch** [19], and **BigVAR** [38] to use machine learning methodologies such as automated parameter tuning, data preprocessing, model blending, cross-validation, performance evaluation, and parallel processing techniques for decreasing model build time.

2 Forecasting Data

2.1 Univariate Forecasting Data

The standard objective in forecasting is, at time period t, make predictions for the target variable y for $t + h$ periods into the future. The difference between standard regression tasks and forecasting is that future values of y are mainly predicted by it's past. This means that forecasting tasks are most suitable when aspects of past patterns will continue on into the future.

Professional forecasters attempt to predict the future of a series based on its past values. Forecasting has applications in a wide range of tasks including forecasting stock prices [25], weather patterns [2], international conficts [9], and earthquakes [47]. This paper uses data from the Makridakis competition To evaluate **mlr**'s forecasting framework as it is well known and provides several types of time series.

The Makridakis competition [35] is a set of forecasting challenges organized by the International Institute of Forecasters and led by Spyros Makridakis to evaluate and compare the accuracy of forecasting methods. The most recent of the competitions, the M4 competition, contains 10,000 time series on a yearly, quarterly, monthly, and daily frequency in areas such as finance, macroeconomics, climate, microeconomics, and industry. A particular climate series is used to example **mlr**'s forecasting features. The data is daily with the training subset starting on September 6th, 2007 and ending on September 5th, 2009 while the testing subset is from September 6th, 2009 to October 10th, 2009 for a total of 640 training periods and 35 test periods to forecast.

```r
# Read in M4 Data
library(M4comp)
library(xts)
library(lubridate)
m4.climate <- M4[[8836]]
m4.train <- xts(m4.climate$past, as.POSIXct("2007-12-05") +
                  days(0:I(length(m4.climate$past)-1)))
m4.test <- xts(m4.climate$future, as.POSIXct("2009-09-06") +
                 days(0:I(length(m4.climate$future)-1)))
colnames(m4.train) <- "target_var"
colnames(m4.test) <- "target_var"
```

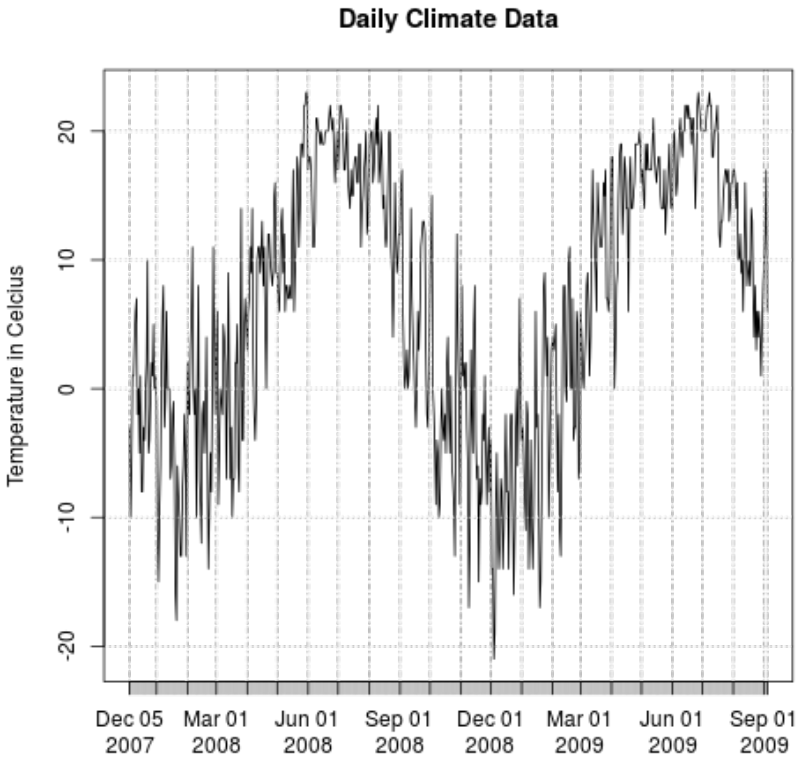


Figure 1: The training data from the M4 forecasting competition

This series has a distinct seasonality feature, which will give the forecasting

tools in **mlr** a challenge. Figure 1 is what most would imagine when they think of a time series. There is a clear seasonal time trend with individual points moving about the annual seasons. The data can be found in the package **M4comp** [3] under sets`M4[8836]`.

2.2 Multivariate Forecasting Data

This paper uses the EUStockMarkets data set from the **datasets** [40] package to showcase the multivariate forecasting tools. It contains a subset of daily DAX, SMI, CAC, and FTSE European stock indices from July 1st, 1991 to August 24th, 1998 totaling 1828 training observations and 32 test observations.

```
# Read in Stock Market Data
data("EuStockMarkets")
EuStockMarkets.time = lubridate::date_decimal(
                as.numeric(time(EuStockMarkets)))
EuStockMarkets  = xts::xts(as.data.frame(EuStockMarkets),
                           order.by = EuStockMarkets.time)
eu.train = EuStockMarkets[1:1828,]
eu.test = EuStockMarkets[1829:1860,]
```

Figure 2 shows the plot of each series. Note that each stock index tends to follow a similar, but diverging, trend. This will be important to note when performing windowed cross-validation as it will be a real test to see how well the models adapt to what appears to be nonstationary data.

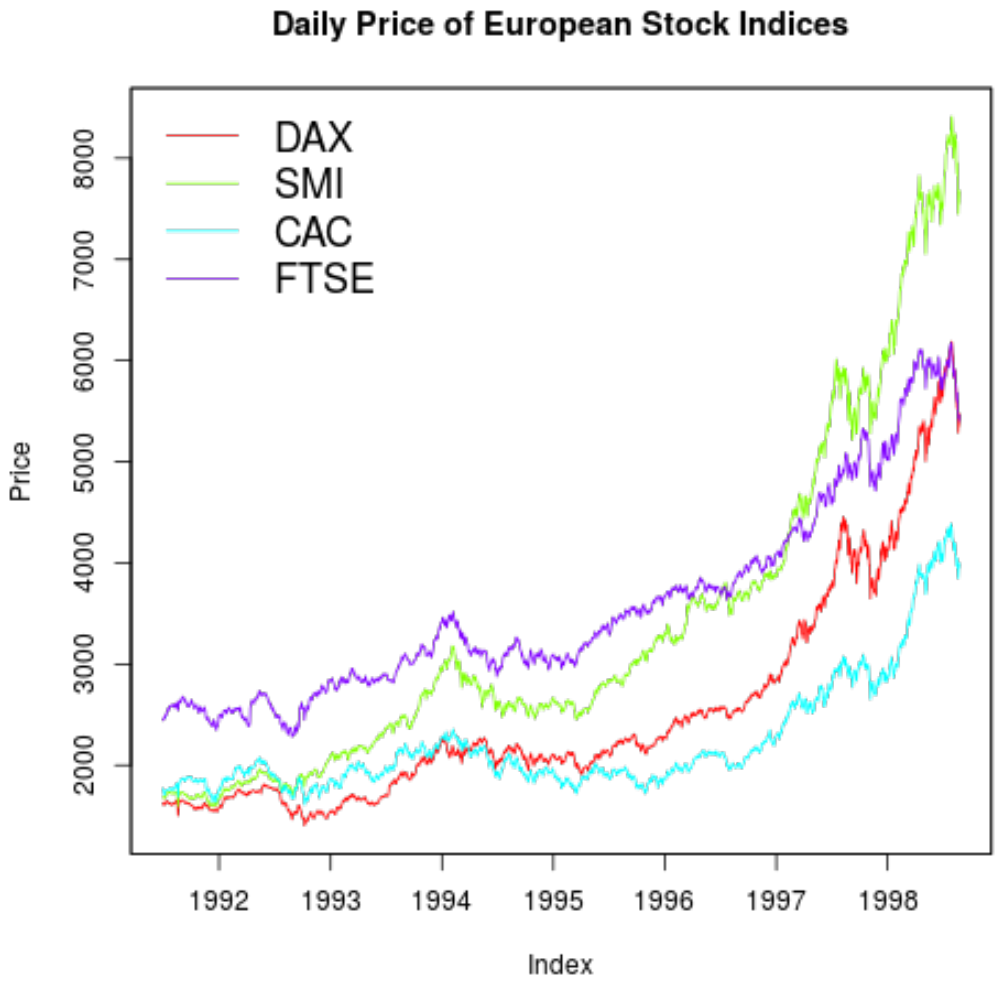


Figure 2: The training data from the EuStockMarkets dataset

3 Univariate and Multivariate Forecasting Tasks

3.1 Univariate Tasks

mlr uses the S3 object system to clearly define a predictive modeling task. Tasks contain the data and other relevant meta-information such as the task

id and for supervised learning problems the target variable. Forecasting tasks are handled in **mlr** by the function `makeForecastRegrTask()`. The forecasting task inherets most of it's arguments from `makeRegrTask`, but has two additional arguments.

data: Instead of a data frame, an xts object from **xts** [45] containing the time series.

frequency: An integer representing the periodicity of the time series. For example, daily data with a weekly periodicity has a frequency of 7, daily data with a yearly periodicity has a frequency of 365, and weekly data with a yearly frequency has a periodicity of 52.

```
library(mlr)

climate.task = makeForecastRegrTask(id = "M4 Climate Data",
                                    data = m4.train,
                                    target = "target_var",
                                    frequency = 183L)
climate.task

## Task: M4 Climate Data
## Type: fcregr
## Target: target_var
## Observations: 640
## Dates:
##  Start: 2007-12-05
##  End:   2009-09-04
## Frequency: 183
## Features:
## numerics  factors  ordered
##        0        0        0
## Missings: FALSE
```

```
## Has weights: FALSE
## Has blocking: FALSE
```

Like a regression task, this records the type of the learning problem and basic information about the data set such as the start and end dates, frequency, and whether there are missing values. Note that there are zero features in the task because there is only a target variable, which the model itself will use to build features.

3.2 Multivariate Tasks

One common problem with forecasting is that it is hard to use additional explanatory variables or forecast multiple targets that are dependent on one another. If a model with exogeneous variables is at time t and is designed to predict ten periods in the future, it needs to know the values of the exogeneous variables at time $t + 10$, which is often not possible. A new set of models [38] which treats exogeneous variables endogenously allows forecasters to not only forecast the target, but additional explanatory variables. Forecasting exogenous variables works by treating all the variables as targets, making them endogenous to the model. A multivariate forecasting task is created to hold the data and meta-information for the model. The function `makeMultiForecastRegrTask()` has the same arguments as `makeForecastRegrTask()` with one exception. The `target` argument can contain either a single target variable, multiple target variables, or `All` which treats all variables endogenously.

```
mfcregr.univar.task = makeMultiForecastRegrTask(id = "bigvar",
                                                data = EuStockMarkets,
                                                target = "FTSE",
                                                frequency = 365L)
mfcregr.univar.task

## Task: bigvar
## Type: mfcregr
## Target: FTSE
## Observations: 1860
## Dates:
##  Start: 1991-07-01 02:18:27
##  End:   1998-08-24 20:18:27
## Frequency: 365
## Features:
## numerics  factors  ordered
##        3        0        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
```

Like `makeForecastRegrTask()`, `mfcregr.univar.task` has the standard output, but notice now that there are three features. Alternatively, `mfcregr.all.task` contains multiple target values with no features. The difference between each of these multivariate tasks is that `mfcregr.univar.task` will act similar to `makeForecastRegrTask()`, giving a univariate forecast output and evaluating the forecasts with univariate measures. When the target is `All` or multiple variables the trained model will forecast and output all series and use a multivariate form of the univariate measures. Though the results appear different, both of these tasks will still forecast all of the underlying series, which allows the model take exogeneous models and treat them endogeneously for n-step ahead forecasts that use additional explanatory variables.

```
mfcregr.all.task = makeMultiForecastRegrTask(id = "bigvar",
                                             data = eu.train,
                                             target = "all",
                                             frequency = 365L)
mfcregr.all.task
```

```
## Task: bigvar
## Type: mfcregr
## Target: DAX SMI CAC FTSE
## Observations: 1828
## Dates:
##  Start: 1991-07-01 02:18:27
##  End:   1998-07-10 22:09:13
## Frequency: 365
## Features:
## numerics  factors  ordered
##        0        0        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
```

4 Building and Tuning a forecast learner

4.1 Univariate Forecasting

The `makeLearner()` function provides a structured model building framework to the several forecasting models currently implemented in **mlr**. The Trigonometric, exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal Components (TBATS) [34] is built to show how to create forecasting models in **mlr**.

TBATS is one of the most popular forecasting models and is available in **mlr** along with models such as BATS, ARIMA, ETS, several GARCH variants,

and autoregressive neural networks. Section 6.1 uses preprocessing features to develop arbitrary supervised machine learning models in the context of forecasting. A TBATS model is created by calling the function `makeLearner()`, supplying the class of learner, order, the number of steps to forecast, and any additional arguments to be passed to `tbats()` for **forecast**.

```
tbats.mod =makeLearner("fcregr.tbats", use.box.cox = TRUE,
                       use.trend = TRUE,
                       seasonal.periods = TRUE, max.p = 60, max.q = 60,
                       stationary = FALSE, use.arma.errors = TRUE,
                       h = 35, predict.type = "response")
```

The predict type for forecasting models can either be a point estimate (`response`) or point estimates with quantiles of confidence intervals (`quantile`). The function `train()` is called, supplying both the learner and task to build the final model over the full data set.

```
train.tbats= train(learner = tbats.mod, task = climate.task )
```

The forecasts are generated by calling `predict()`. Optionally, supplying the test data as an additional argument will allow **mlr** to return an object containing meta information for the forecasts along with the prediction and test data in columns `truth` and `response`, respectively.

```
predict.tbats = predict(train.tbats, newdata = m4.test)
predict.tbats

## Prediction: 35 observations
## predict.type: response
```

```
## threshold:
## time: 0.00
##            truth response
## 2009-09-06     7 3.231406
## 2009-09-07    12 4.425507
## 2009-09-08    14 5.380601
## 2009-09-09    10 5.617823
## 2009-09-10    10 5.511217
## 2009-09-11    16 5.331298
## ... (35 rows, 2 cols)
```

The model is evaluated by calling `performance()` with the Mean Absolute Scaled Error (MASE) [29] measure.

```
performance(predict.tbats, measure = mase,
            task = climate.task)

##      mase
## 0.0676601
```

MASE has favorable properties for calculating forecast errors relative to measures such as root mean squared error or median relative absolute error. Arguably one of the most important features, it is very interpretable. Let y_t and $\tilde{y}_t$ be the target variable and prediction at time t with $\epsilon_t = y_t - \tilde{y}_t$ being the forecast error. Then MASE is calculated as

$$\text{MASE} = \frac{\sum_{t=1}^{T} |\epsilon_t|}{\frac{T}{T-1} \sum_{t=2}^{T} |y_{t,\text{insample}} - y_{t-1,\text{insample}}|} \tag{1}$$

Where the denominator is the one step ahead naive forecast from the training data. When the numerator is equal to the denominator, the model performs as well as a simple naive forecast method. A MASE score greater

than one indicates the model is performing worse than the naive forecasting method while scores less than one mean the model is performing better than the naive forecasting method.

The scale invariance of MASE means that it is independent of the scale of the data and allows model comparison across data sets. The scale invariance of MASE has made it a favorite for comparing the accuracy of forecast methods [16] across datasets. While scaling in measures such as the Mean Absolute Percentage Error can cause poor behavior as the target variable goes to zero, MASE does not become skewed when the target variable approaches zero. Having good properties near zero allows the use of MASE in situations in which zeros occur frequently or zero is not meaningful such as predicting temperature.

The model does a successful job of catching the downward trend, but figure 3 shows the downward bias of the model which does not allow it to account for the sharp upward swings.

4.2 Updating Forecast Models

Forecasting models are designed to predict the next n values that will appear in the series. Without a way to update the model with new data, the model will have to be completely rebuilt for each new set of forecasts. The `updateModel` function allows forecasters to avoid the expensive cost of retraining the model. The function updates the model with new points which then allows for updated forecasts.

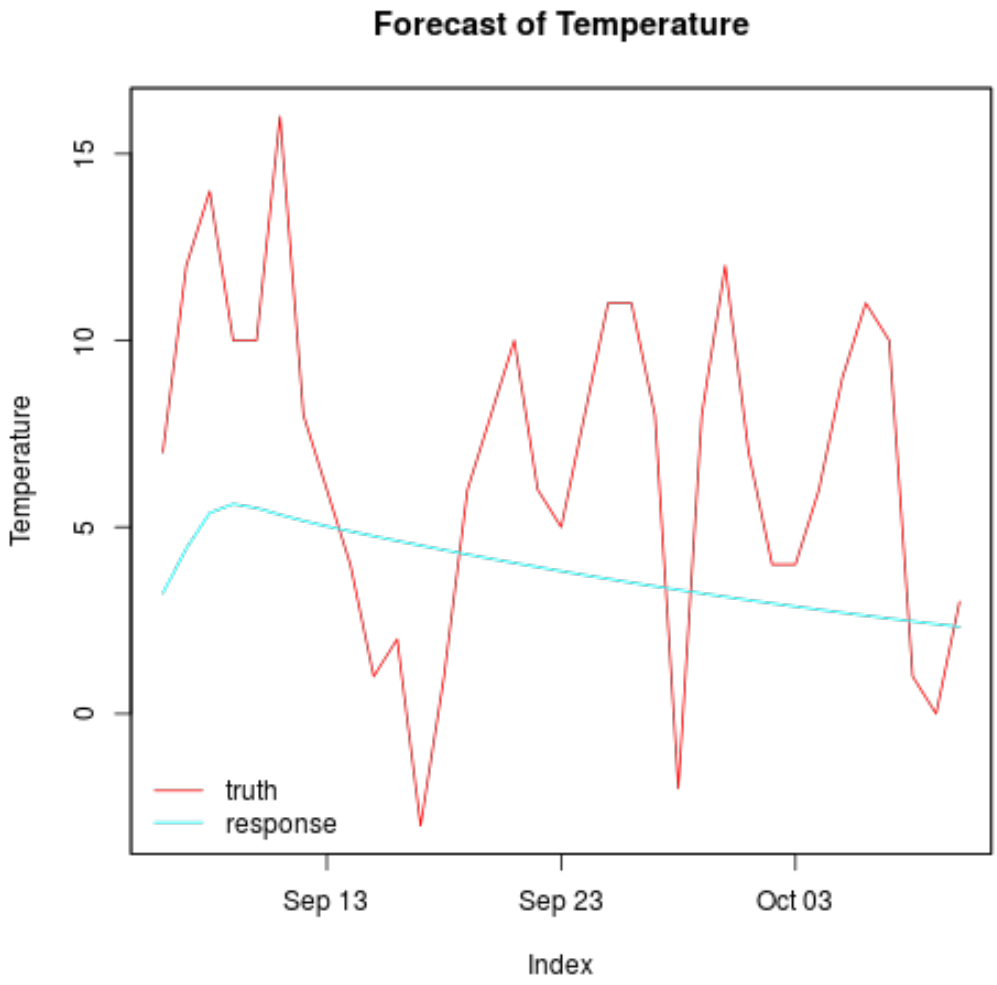


Figure 3: The forecasts response from TBATS against the true testing weather data

```
update.tbats = updateModel(train.tbats, climate.task,
                           newdata = m4.test)
predict(update.tbats, task = climate.task)

## Prediction: 35 observations
## predict.type: response
## threshold:
## time: 0.00
```

```
##                      response
## 2009-09-04 23:59:54 6.319959
## 2009-09-05 23:59:48 7.875401
## 2009-09-06 23:59:43 8.209328
## 2009-09-07 23:59:37 8.157638
## 2009-09-08 23:59:31 8.083591
## 2009-09-09 23:59:26 8.055280
## ... (35 rows, 1 cols)
```

All univariate forecasting models have access to the `updateModel()` function. Future versions of `mlr` will also include online machine learning models that can be updated as new data comes in.

4.3 Multivariate Forecasting

Multivariate forecasting in **mlr** uses the package **BigVAR** [37]. **BigVAR** allows for estimation of high dimensional time series through methods such as the structured Lasso penalties method that finds the best autoregressive structure through cross-sections and time.

```
bigvar.mod = makeLearner("mfcregr.BigVAR",p = 25, struct = "SparseLag",
                         gran = c(50, 60),h = 35, n.ahead = 35)
```

This section uses the multivariate forecast task which has all variables as targets, while section 8.2 performs the single target multivariate forecast with a stacked predictor. Multiforecast regressions operate the same as other supervised models, supplying a task and learner to `train()`.

```
train.bigvar = train(learner = bigvar.mod, task = mfcregr.all.task )
train.bigvar
```

```
## Model for learner.id=mfcregr.BigVAR; learner.class=mfcregr.BigVAR
## Trained on: task.id = bigvar; obs = 1828; features = 0
## Hyperparameters: p=25,struct=SparseLag,gran=50,60,h=35,n.ahead=35
```

Predictions for `multiForecast` methods have a similar output to `multiclass` methods, returning multiple truth and response variables. For multivariate forecasts, a multivariate version of MASE takes the mean of each MASE score for the individual variables as the performance measure.

$$\text{MultiMASE} = \frac{\sum_{i=1}^{m} \text{MASE}_i}{m} \tag{2}$$

given that m is the number of variables that are forecast.

```
predict.bigvar = predict(train.bigvar, newdata = eu.test)
predict.bigvar
```

```
## Prediction: 32 observations
## predict.type: response
## threshold:
## time: 0.02
##                     truth.DAX truth.SMI truth.CAC truth.FTSE response.DAX
## 1998-07-12 07:50:46   5905.15    8047.3    4252.1     5960.2     5852.841
## 1998-07-13 17:32:18   5961.45    8099.0    4304.4     5988.4     5820.194
## 1998-07-15 03:13:50   5942.06    8166.0    4311.1     5990.3     5801.712
## 1998-07-16 12:55:23   5975.88    8160.0    4333.1     6003.4     5790.565
## 1998-07-17 22:36:55   6018.89    8227.2    4339.9     6009.6     5781.540
## 1998-07-19 08:18:27   6000.84    8205.0    4319.2     5969.7     5772.087
##                     response.SMI response.CAC response.FTSE
## 1998-07-12 07:50:46     7982.503     4192.149      5997.825
## 1998-07-13 17:32:18     7960.779     4153.550      6037.958
## 1998-07-15 03:13:50     7955.264     4130.674      6063.687
## 1998-07-16 12:55:23     7958.119     4115.612      6083.094
## 1998-07-17 22:36:55     7965.061     4103.302      6100.075
## 1998-07-19 08:18:27     7973.586     4091.154      6116.419
## ... (32 rows, 8 cols)
```

```
performance(predict.bigvar, multivar.mase, task = mfcregr.all.task)

## multivar.mase
##     0.2209703
```

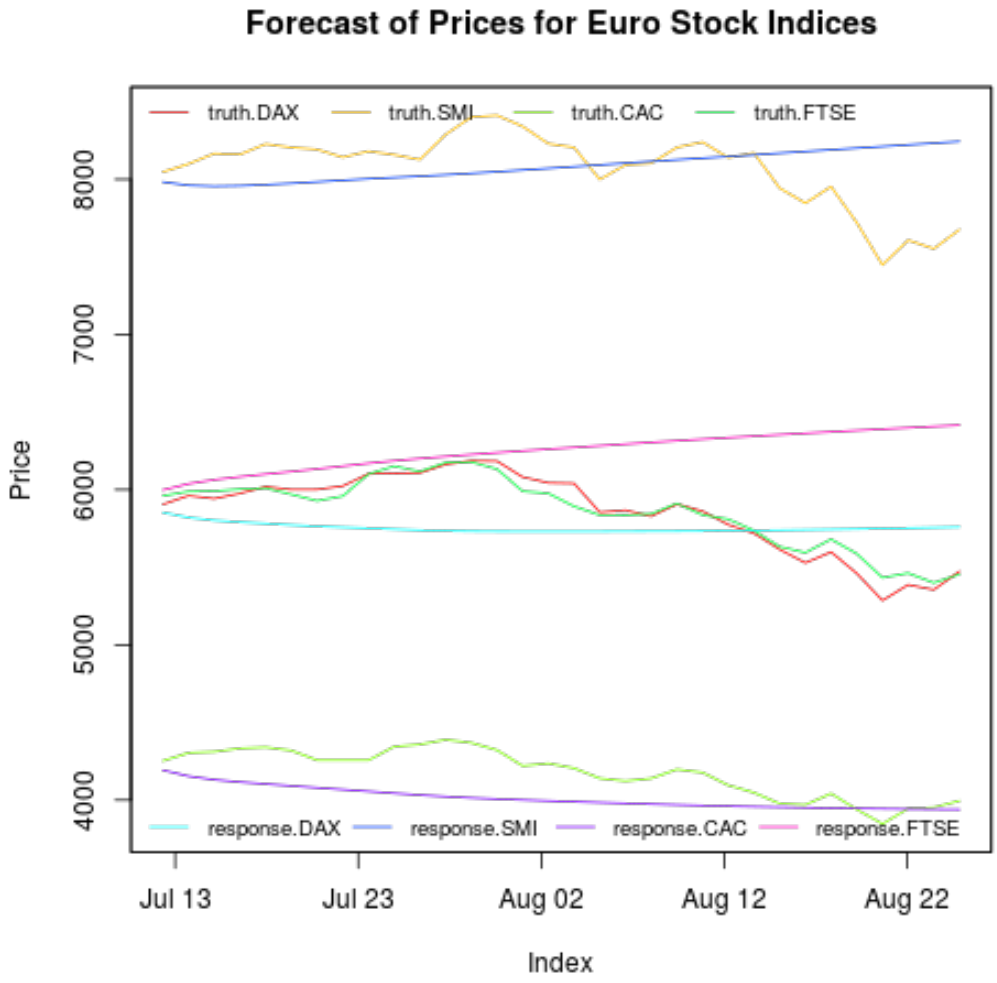


Figure 4: The forecasts response from BigVAR against the true test stock index data

5 Resampling with Time

While TBATS is one of the most popular time series models, the order selection process of the ARIMA errors and whether to include trend, damped trend, or seasonal periods can be a subjective process that makes finding the best model difficult for users. One of the first proposals for automated forecasting methods comes from [13] for automatic order selection of ARIMA models. Innovations are obtained by fitting high order autoregressive models to the data and then computing the likelihood of potential models through a series of standard regression. Proprietary algorithms from software such as Forecast Pro [43] and Autobox [42] are well known and have performed to high standards in competitions such as the M3 forecasting competition [35]. One of the most prominent R packages for automated forecast is **forecast** [28] which contains several methods for automated forecasting including exponential smoothing based methods and step-wise algorithms for finding optimal ARIMA models.

Forecasting in **mlr** takes a machine learning approach, creating a parameter set for a given model and using an optimization method to search over the parameter space. The forecasting extension of **mlr** includes growing and fixed window resampling schemes to train over the possible models. Resampling schemes such as cross-validation and bootstrapping are common in machine learning for dealing with the bias-variance tradeoff [17] [44]. When there is a time component to the data, windowing schemes are useful in allowing a valid resampling scheme while still accounting for the time properties of the series. Figure 5 gives an example of fixed and growing window cross validation. Let h be a forecast horizon where the index of the starting and end points of the

training data are start_i and end_i where i is the index for each window. The first model will train on data between start_i and end_i while testing on data indexed between $\text{end}_i + h$. After each iteration the window slides forward h steps such that $\text{end}_{i+1} = \text{end}_i + h$. In the case of the fixed window, the starting index will also shift such that $\text{start}_{i+1} = \text{start}_i + h$ while start_i will remain constant in the growing window case. This type of growing and fixed window resampling [27] are now available in the `resampling()` function of **mlr**.

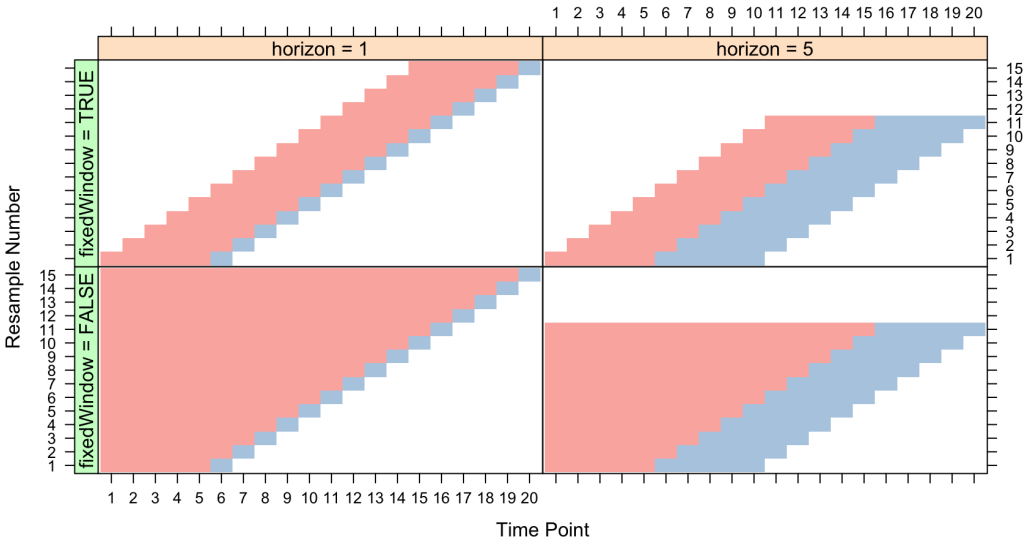


Figure 5: Resampling with a window scheme as exampled by **caret** [30]. The top graphs are fixed window cross validation while the bottom graphs are growing window cross validation.

To create a windowing resampling process, the function `makeResampleDesc()` uses the resampling type, the horizon, initial window, the length of the series, and an optional argument to skip over some windows for the sake of time.

```
resampDesc = makeResampleDesc("GrowingCV", horizon = 35L,
                              initial.window = .7,
                              size = nrow(getTaskData(climate.task)),
                              skip = .004)
resampDesc

## Window description:
##  growing with 79 iterations:
##  448 observations in initial window and 35 horizon.
## Predict: test
## Stratification: FALSE
```

The methods in package **ParamHelpers** [6] allow **mlr** to have a simple and rigorous interface for creating parameter spaces. There are several types of tools to help search the parameter space including grid search, random search [4], iterated F-racing, Simulated Annealing, and CMA-ES [1] to search the parameter space for the most optimal model.

```
parSet = makeParamSet(
           makeLogicalParam(id = "use.box.cox",default = FALSE,
                            tunable = TRUE),
           makeLogicalParam(id = "use.trend", default = FALSE,
                            tunable = TRUE),
           makeLogicalParam(id = "use.damped.trend", default = FALSE,
                            tunable = TRUE),
           makeLogicalParam(id = "seasonal.periods", default = FALSE,
                            tunable = TRUE),
           makeIntegerParam(id = "max.p", upper = 30, lower = 1,
                            trafo = function(x) x*2),
           makeIntegerParam(id = "start.p", upper = 30, lower = 1,
                            trafo = function(x) x*2),
           makeIntegerParam(id = "max.q", upper = 30, lower = 1,
                            trafo = function(x) x*2),
           makeIntegerParam(id = "start.q", upper = 5, lower = 0,
                            trafo = function(x) x*2),
```

```
            makeIntegerParam("max.P", lower = 0, upper = 5),
            makeIntegerParam("max.Q", lower = 0, upper = 5),
            makeDiscreteParam("ic",values = c("aicc","aic","bic")),
            makeDiscreteParam("test",values = c("kpss","adf","pp")),
            makeDiscreteParam("seasonal.test",
                              values = c("ocsb", "ch")),
          makeLogicalParam("biasadj", default = FALSE)
        )

#Specify tune by grid estimation
ctrl = makeTuneControlIrace(maxExperiments = 500L)
```

Using `tuneParams()` the model is tuned for the task using the specified resampling scheme, parameter set, tune control, and measure. The `trafo` argument allows the tuning process to take values on a separate scale than the one described. In the tuning scheme above, trafo will search over $1, 4, 9, ..., 81$, and 100 values of lag, selecting the model with the best lag structure. This tuning task uses MASE [29] as a measure of performance [1]. The model is also tuned in parallel using the package **parallelMap** [5] for it's built in compatibility with **mlr**.

```
#Tune the TBATS Model
library("parallelMap")
parallelStartSocket(8)
configureMlr(on.learner.error = "warn")
set.seed(1234)
tbatsTune = tuneParams(makeLearner("fcregr.tbats", h = 35),
                       task = climate.task,
                       resampling = resampDesc, par.set = parSet,
                       control = ctrl, measures = mase)
parallelStop()
```

[1] Models with a seasonal difference > 0 may be favorably biased as **mlr**'s forecasting extension uses the non-seasonal MASE score

```
# Output the test mean of best model
tbatsTune$y
```

```
## mase.test.mean
##     0.06195313
```

Using **mlr**'s built in plotting routines the tuning parameters can be analyzed graphically using partial dependence plots [22]. Because of constraints which do not allow logical types in the partial dependence plots, the below code takes all of the logical parameters from `tbatsTune` and coerces them to be numeric.

```
# Making graphics
plotHyperParsEffect(tbats.hyp, x= "max.q", y = "mase.test.mean",
                    plot.type = "line",
                    partial.dep.learn = "regr.randomForest")
plotHyperParsEffect(tbats.hyp, x= "max.P", y = "mase.test.mean",
                    plot.type = "line",
                    partial.dep.learn = "regr.randomForest")
```

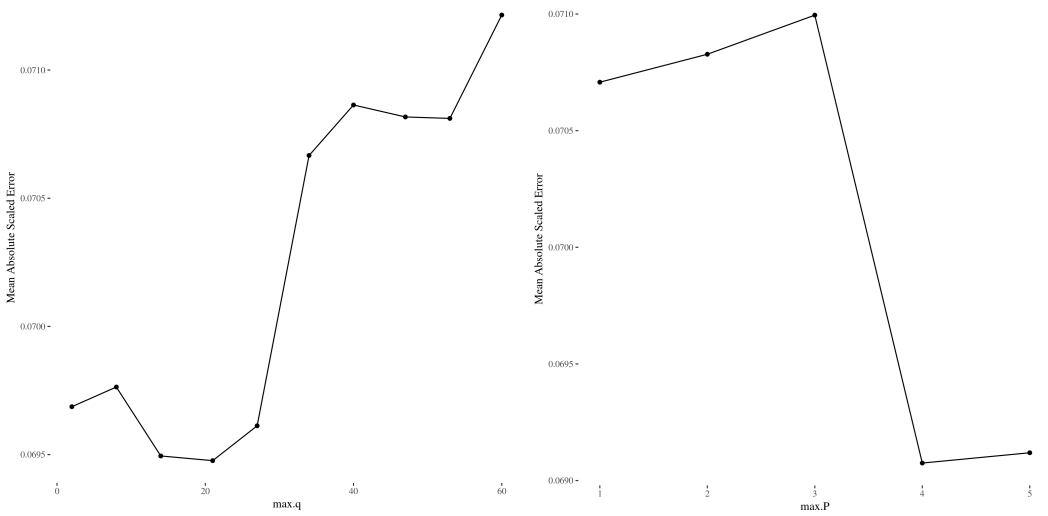


Figure 6: The figure on the left shows the dependence of the MASE score averaged over all windows for each value of the maximum number of moving average terms q. The figure on the right does the same, but for the autoregressive term p

For this model the best maximum number of moving average coefficients is two while the best number of seasonal autoregressive terms is five. The best model's parameters are extracted using `setHyperPars()` and passed to `train()` to go over the full data set.

```
# Train best model over full data
lrn = setHyperPars(makeLearner("fcregr.tbats", h = 35),
                   par.vals = tbatsTune$x)
m = train(lrn, climate.task)
```

To make predictions the task test data are passed to `predict()`

```
climate.pred = predict(m, newdata = m4.test)
performance(climate.pred, measures = mase, task = climate.task)

##      mase
## 0.0562274
```

For comparison, TBATS is run in **forecast** with it's base parameters

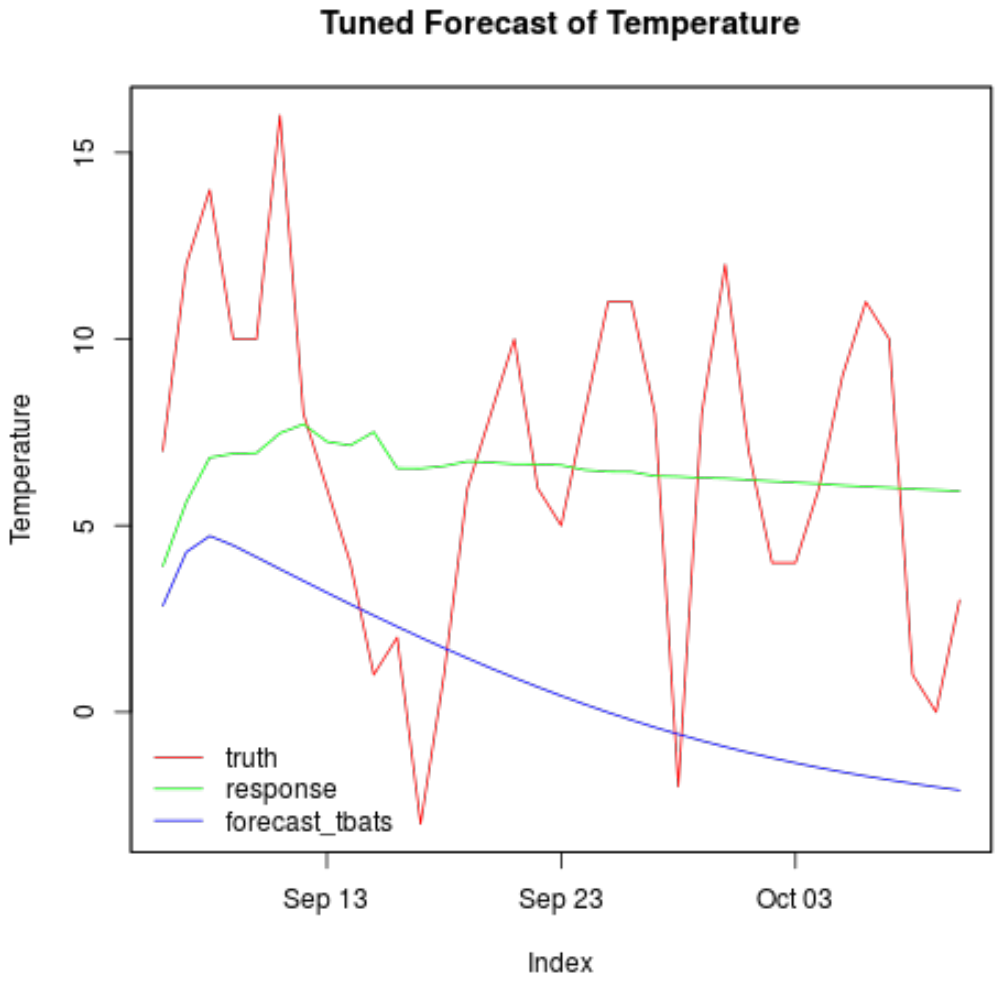
```
library(forecast)
forecast.train.tbats = tbats(ts(m4.train,frequency = 365))
forecast.tbats = forecast(forecast.train.tbats,h=35)
# Calculate MASE
sum(abs(coredata(m4.test)-forecast.tbats$mean)) /
 ((nrow(m4.test) / (nrow(m4.test) + 1)) *
     sum(abs(diff(coredata(m4.test)))))

## [1] 1.951852
```

Now it is possible to see the clear benefit of the forecast extension to **mlr**. While **forecast** does some automatic tuning, this model would still require

extensive manual testing in an attempt to find the best model. The automation of the modeling process allows for a faster and better solution with minimal work.

Figure 7: The TBATS model forecasts tuned in **mlr** Vs. the base TBATS model in forecast



To this author's knowledge, this is the first package in R that allows for

automated tuning and training of GARCH models [15]. It is possible to pass and train multiple types of GARCH models while also tuning the model's respective parameters. In this example `predict.type = "quantile"` is used to estimate confidence intervals for the forecast.

```
# Make a tuning grid for GARCH
par_set = makeParamSet(
  makeDiscreteParam(id = "model", values = c("sGARCH", "csGARCH")),
  makeIntegerVectorParam(id = "garchOrder", len = 2L, lower = c(1,1),
                         upper = c(4,4)),
  makeIntegerVectorParam(id = "armaOrder", len = 2L, lower = c(5,1),
                         upper = c(8,3)),
  makeLogicalParam(id = "include.mean"),
  makeLogicalParam(id = "archm"),
  makeDiscreteParam(id = "distribution.model",
                    values = c("norm","std","jsu")),
  makeDiscreteParam(id = "stationarity", c(0,1)),
  makeDiscreteParam(id = "fixed.se", c(0,1)),
  makeDiscreteParam(id = "solver", values = "nloptr")
)

#Specify tune by F-racing
ctrl = makeTuneControlIrace(maxExperiments = 400L)

#Tuning in parallel
parallelStartSocket(6)
configureMlr(on.learner.error = "warn")
set.seed(1234)
garchTune = tuneParams(makeLearner("fcregr.garch", n.ahead= 35),
                       task = climate.task, resampling = resampDesc,
                       par.set = par_set, control = ctrl,
                       measures = mase)
parallelStop()
garchTune$y
```

```
## mase.test.mean

##     0.07888857
```

Because the partial dependence plots are still in development they do not work well with `makeNumericVectorParams()` such as `garchOrder` and `armaOrder`. Instead of looking at each p, q individually, the sum of the coefficient values for the p, q order of the GARCH component and the ARMA component are plotted.

```
# Make hyper parameter data
garch.hyperpar = generateHyperParsEffectData(garchTune,
                     trafo = TRUE, include.diagnostics = FALSE,
                     partial.dep = TRUE)
garch.hyperpar$data = garch.hyperpar$data[
                       -which(garch.hyperpar$data$mase.test.mean ==
                             max(garch.hyperpar$data$mase.test.mean)),]
garch.hyperpar$data$garchOrder1 = garch.hyperpar$data$garchOrder1 +
                                  garch.hyperpar$data$garchOrder2
garch.hyperpar$data$garchOrder2 = NULL
colnames(garch.hyperpar$data)[2] = "garchOrder"
garch.hyperpar$data$armaOrder1 = garch.hyperpar$data$armaOrder1 +
                                 garch.hyperpar$data$armaOrder2
garch.hyperpar$data$armaOrder2 = NULL
colnames(garch.hyperpar$data)[3] = "armaOrder"
garch.hyperpar$data$include.mean = as.numeric(garch.hyperpar$data$include.mean)
garch.hyperpar$data$archm = as.numeric(garch.hyperpar$data$archm)
```

```
plotHyperParsEffect(garch.hyperpar, x= "garchOrder", y = "mase.test.mean",
                     plot.type = "line",
                     partial.dep.learn = "regr.randomForest")
plotHyperParsEffect(garch.hyperpar, x= "armaOrder", y = "mase.test.mean",
                     plot.type = "line",
                     partial.dep.learn = "regr.randomForest")
```

```
## Saving 7 x 7 in image
## Saving 7 x 7 in image
```

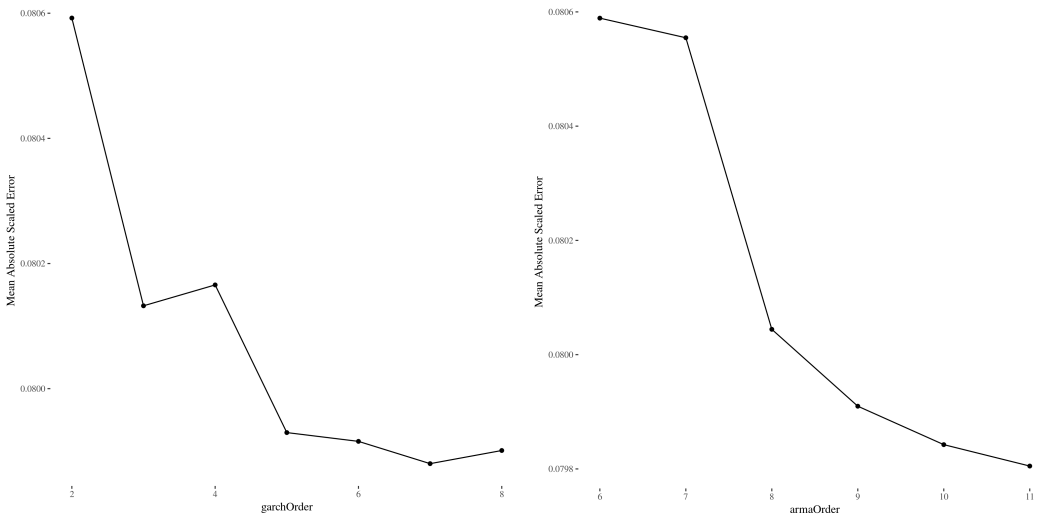


Figure 8: The figure on the left shows the dependence of the MASE score averaged over all windows for each value of the maximum number of p, q terms in the GARCH order. The figure on the right does the same, but for the ARMA order

The dependence plots show that for both the GARCH and ARMA orders, as the order increases, the MASE score decreases. A better model would arise if the order of both the GARCH and ARMA components increased up to twenty or more. The forecasting extension of **mlr** treats forecasting models with quantiles the same as any other learner post-tuning. The best hyperparameters are taken, and a final model trains over the entire dataset.

```
# Get the hyperpars and train the best GARCH model
tuned.lrn = setHyperPars(makeLearner("fcregr.garch",
                                     predict.type = "quantile"),
```

```
                          par.vals = garchTune$x)
garch.train = train(tuned.lrn, climate.task)
```

```
climate.pred = predict(garch.train, newdata = m4.test)
climate.pred

## Prediction: 35 observations
## predict.type: quantile
## threshold:
## time: 0.01
##            truth response se.quantile0.05 se.quantile0.95
## 2009-09-06     7 3.870648      -2.8377519        10.57905
## 2009-09-07    12 5.940932      -0.7674667        12.64933
## 2009-09-08    14 6.962520       0.1060161        13.81902
## 2009-09-09    10 7.336334       0.3367573        14.33591
## 2009-09-10    10 7.473346       0.4736223        14.47307
## 2009-09-11    16 7.737601       0.7377337        14.73747
## ... (35 rows, 4 cols)
```

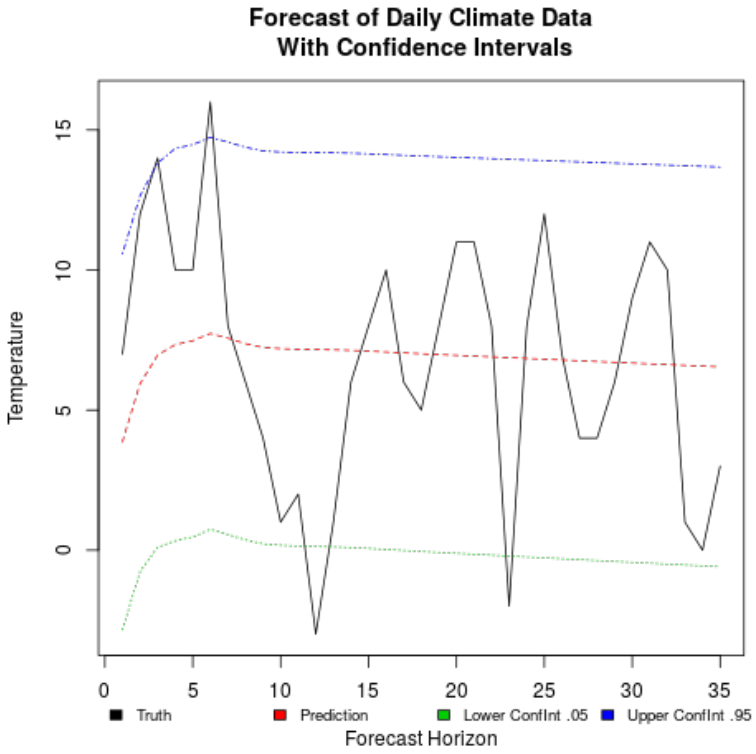


Figure 9: The GARCH model with confidence intervals

```
performance(climate.pred, measures = mase, task = climate.task)

##       mase
## 0.05589035
```

The simple model here performs relatively well. The extremes of the test data are almost all within the given bounds of the 95% confidence interval. With a bit more tuning and perhaps an alternative control for the search over the parameter space, the reader would easily find a better model. While GARCH is excellent at catching the conditional heteroscedasticity of the past series, the TBATS model performs well its ability to catch long or multiseasonal

type periods. Section 8.1 goes over how to stack forecasting models to make use of the best characteristics of each.

6 Forecasting with Machine Learning Models

6.1 Forecasting with Regression Tasks

The forecasting extension of **mlr** includes a preprocessing function that allows forcasting with supervised machine learning models. the function `createLagDiffFeatures()` allows for $AR(p, d)$ structures to be imbedded in machine learning models.

```
climate.regr.task = makeRegrTask(id = "lagged gbm",
                                 data = as.data.frame(m4.train),
                                 target = "target_var")
climate.task.lag = createLagDiffFeatures(climate.regr.task,
                                         lag = 1L:24L,
                                         difference = 1L,
                                         na.pad=FALSE)
climate.task.lag

## Supervised task: lagged gbm
## Type: regr
## Target: target_var
## Observations: 615
## Features:
## numerics  factors  ordered
##       24        0        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
```

Notice that `createLagDiffFeatures()` returns a new task with the lagged variables as the new features. Once the lagged task is created the model is trained or tuned like any other.

```
# make Gradient Boosting Machine
lag.gbm = makeLearner("regr.gbm", par.vals = list(n.trees = 20000,
                                                  shrinkage = .000001,
                                                  interaction.depth = 15,
                                                  bag.fraction = .7))
gbm.train = train(lag.gbm, climate.task.lag)
```

The `forecast()` function allows machine learning models to do arbitary n-step ahead forecasts. Let the one step ahead forecast be defined by

$$\hat{y}_{t+1} = \sum_{i=1}^{p} (\rho_i \Delta_d y_i + \epsilon_i) \tag{3}$$

where ρ_i is the autoregressive parameter of order p and d is the lag of the difference operator Δ. Then the n-step ahead forecast is defined as

$$\hat{y}_{t+n} = \sum_{i=t+1}^{n} (\rho_i \Delta_d \hat{y}_i + \epsilon_i) \tag{4}$$

```
# Forecast with GBM
gbm.forecast = forecast(gbm.train, h = 35L,
                        newdata = as.data.frame(m4.test))
```

```
gbm.forecast

## Prediction: 35 observations
## predict.type: response
## threshold:
## time: 7.30
##   truth response
## 1     7 7.655469
## 2    12 8.234003
## 3    14 8.234003
```

```
## 4    10 8.234003
## 5    10 8.234003
## 6    16 8.234003
## ... (35 rows, 2 cols)

performance(pred = gbm.forecast,
            measures = mase,
            task = climate.regr.task)

##      mase
## 0.0558791
```

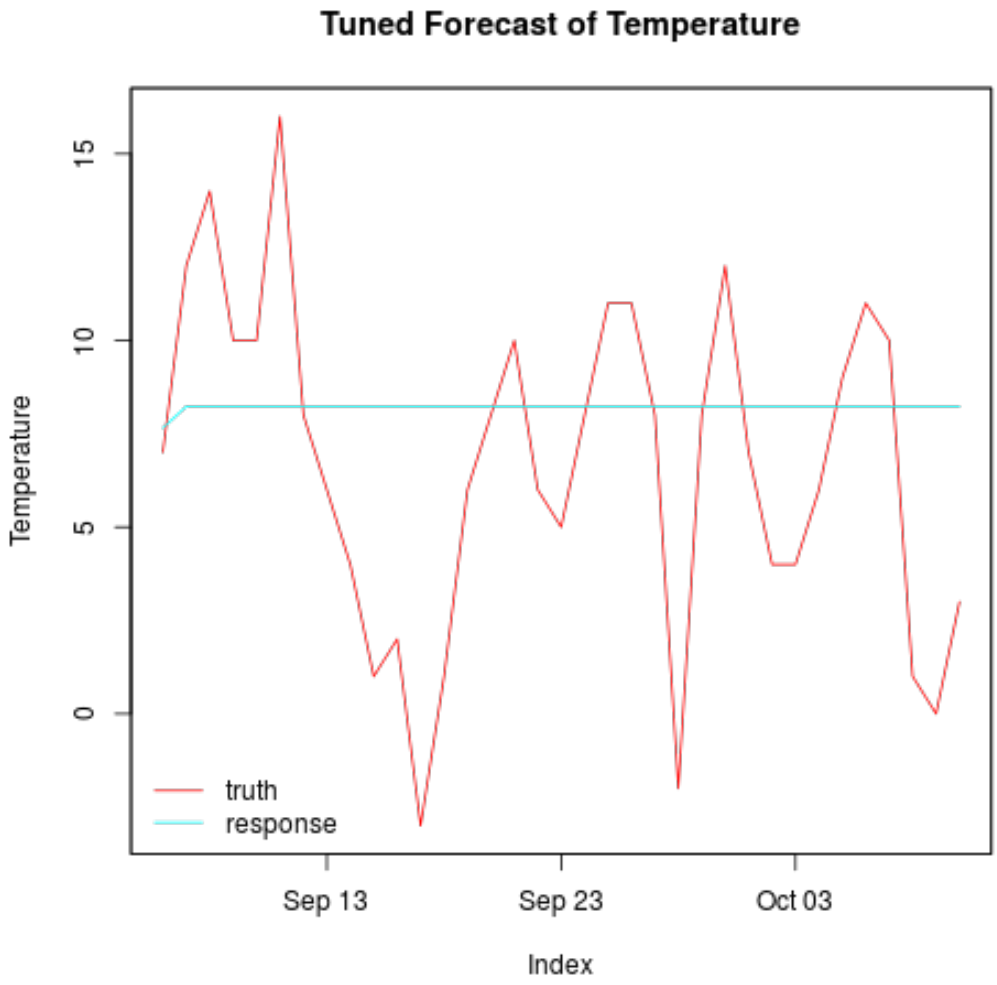


Figure 10: The forecast of the M4 data using a Gradient Boosting Machine

The forecasts drift quickly to an average value, which may be due to the model placing the highest value on the most recent data. When the latest data is much more important than other past periods, this creates a weak decay scheme which causes forecasts to drift quickly to an average value.

6.2 Forecasting with Classification Tasks

Forecasting for binary or multiclass outcomes [31] is a common problem in the real world. However, research in this area of forecasting only started picking up speed in the last decade [14]. The forecasting extension of **mlr** gives econometricians the ability to take all the classification models in **mlr** and apply them to the forecasting context. For developing trading strategies, there are usually a discrete set of choices such as to buy, sell, or hold onto a stock. The forecasting extension of **mlr** allows for classification models that forecast these options [32]. The code below implements a simple buy, sell, or hold trading strategy. If the stock goes up by 5% in a day a buy is executed, down 5% a sell is executed, and otherwise no action will be taken.

```
# Make Trading Strategy
DAX = EuStockMarkets$DAX/lag(EuStockMarkets$DAX,
                             7,nap.pad = FALSE) - 1
trade.strat = ifelse(DAX > .05, "Buy",
                     ifelse(DAX < -.05, "Sell", "Hold"))
trade.strat = trade.strat[8:1860]
euro.classif.data = data.frame(trade.strat = trade.strat ,
                               row.names = index(trade.strat))
euro.classif.train = euro.classif.data[1:1838,,drop = FALSE]
euro.classif.test  = euro.classif.data[1839:1853,,drop = FALSE]
classif.task = makeClassifTask(data = euro.classif.train,
                               target = "DAX")

# Make Lagged Task and Learner
classif.task.lag = createLagDiffFeatures(classif.task,
                                         lag = 1L:565L,
                                         na.pad = FALSE)
classif.learn = makeLearner("classif.boosting", xval = 1,
                            mfinal = 200, minsplit = 10)
```

```
classif.train = train(classif.learn, classif.task.lag)
classif.fc = forecast(classif.train, h=15, newdata = euro.classif.test)
classif.fc
performance(classif.fc)
```

```
## Prediction: 15 observations
## predict.type: response
## threshold:
## time: 422.87
##   truth response
## 1  Hold     Hold
## 2  Hold     Hold
## 3  Sell     Hold
## 4  Hold     Hold
## 5  Hold     Hold
## 6  Hold     Hold
## ... (15 rows, 2 cols)
##      mmce
## 0.4666667
```

With the models and methodologies available in **mlr**, forecasting binary outcomes is now as simple as any other model. These tools can allow for further research in areas such as directions of stock movement [33] and forecasting extreme values [10].

7 Lambert W Transforms

Many machine learning and time series models rely on the assumption that the data or errors fit a normal distribution. This assumption becomes precarious when modeling the asymmetric and fat-tailed data of the real world. Lambert $W \times F()$ transforms are a family of generalized skewed distributions [20] that

have bijective and parametric functions that allow heavy tailed and asymmetric data to appear more Gaussian [21].

Let U be a continuous random variable with cdf $F_U(u|\beta)$ and pdf $f_U(u|\beta)$ given β is a parameter vector. Define a continuous location-scale random variable $X \sim F_X(x|\beta)$. A locaton-scale skewed Lambert $W \times F_X$ random variable is defined as

$$Z = U \exp\left(\frac{\delta}{2}(U^2)\right),\ \delta \geq 0 \tag{5}$$

And the heavy-tailed Lambert $W \times F_X$ random variable can be defined as

$$Z = U \exp\left(\frac{\delta}{2}(U^2)^{\alpha}\right),\ \delta \geq 0\ \alpha > 0 \tag{6}$$

Given that $U = (X - \mu_X)/\sigma_X$ where μ_X, σ_X, δ, and α are the mean and standard deviation of X and the parameters to control skewness and asymetry , respectively. When $\delta = 0$, equation 5 reduces to a standard normal distribution. Equation 5 is the general form of Tukey's h distribution [26] and the basis for Morgenthaler and Tukey's [36] skewed, heavy tailed family of hh random variables.

$$Z = \begin{cases} U \exp\left(\frac{\delta_l}{2}(U^2)_l^{\alpha}\right),\ \delta_l \geq 0\ \alpha_l > 0 \\ U \exp\left(\frac{\delta_r}{2}(U^2)_r^{\alpha}\right),\ \delta_r \geq 0 \alpha_r > 0 \end{cases} \tag{7}$$

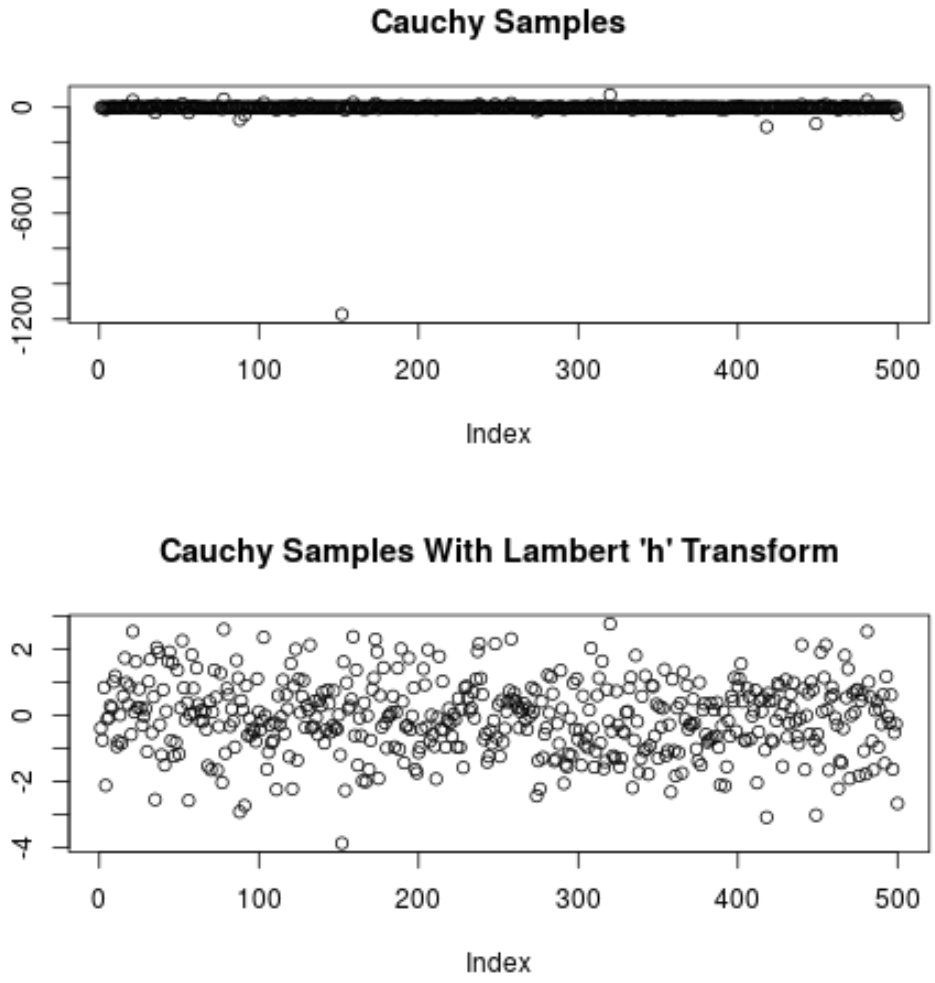


Figure 11: The top plot are samples from the cauchy distribution where the bottom plot represents the cauchy transformed variables with the hh distribution to make the samples approximately normal.

The function `Gaussianize()` is available in the package **LambertW** and has been made into a preprocessing function in **mlr**. Instead of calling `makeLearner()` to create a model, the function `makePreprocWrapperLambert()` can be used to create the model. This means the pre-processing scheme in-

tegrates into the model. Integrating the pre-processing scheme directly on top of the model stops users from accidentally biasing their models due to improperly applying pre-processing processes. For instance, if a user demeaned their entire data set and then split the data into train and test subsets, the training data will be biased because demeaning the model over both the train and test data gives the training data information about the mean of the test data. What should happen instead is, the user first splits the data into training and test data, and then demeans each separately. Similarly, cross-validation follows the above scheme whereby each cross-validation will be biased if the user demeaned all of the training data at one time. It is very easy for a user, who assumes they have made a good faith attempt not to bias their model, to receive overconfident results. Embedding the pre-processing itself allows **mlr** to overcome this.

The pre-processing scheme is broken down into two main components. One function to gather and estimate necessary parameters in the training data, then a function to apply the learned preprocess over the training and test data. In the context of Lambert $W \times F()$ transforms, the estimates for the parameters of the h, hh, or s distribution that gaussianizes the data comes from the training data. Then the estimated parameter values from the training set are used during prediction to gaussianize the test observations. The code below follows this methodology, creating the model with Lambert preprocessing, training the model, and then performing prediction. The result to the user appears the same, but there is a significant reduction of bias in the background.

```
# Make a Learner with Lambert WxF() learner
lamb.lrn = makePreprocWrapperLambert("classif.lda", type = "h")
lamb.lrn

## Learner classif.lda.preproc from package MASS
## Type: classif
## Name: ; Short name:
## Class: PreprocWrapperLambert
## Properties: numerics,factors,prob,twoclass,multiclass
## Predict-Type: response
## Hyperparameters: type=h,methods=IGMM,verbose=FALSE

lamb.trn = train(lamb.lrn,iris.task, subset = 1:120)
lamb.pred = predict(lamb.trn, iris.task, subset = 121:150)

# Do the non-LW version
trn = train(makeLearner("classif.lda"),iris.task, subset = 1:120)
pred = predict(trn, iris.task, subset = 121:150)
performance(lamb.pred)

## mmce
##  0.1

performance(pred)

## mmce
##  0.1
```

8 Stacking Forecasting Learners

Stacking is a form of ensemble learning [12] in which a learning algorithm trains on the predictions of several other learning algorithms. Let $y_{i,m}$ be the prediction at time i of model m. Given an aggregation function ϕ, a stacked forecast learner [11] is represented as

$$\tilde{y}_{i+1} = \phi(\tilde{y}_{i+1,1}, \tilde{y}_{i+1,2}, \ldots, \tilde{y}_{i+1,m}, \sum_{j=1}^{m} \epsilon_{i+1,j}) \tag{8}$$

For a simple $\phi()$ such as the ensemble average, equation 8 becomes

$$\tilde{y}_{i+1} = \frac{\tilde{y}_{i+1,1} + \tilde{y}_{i+1,2} + \cdots + \tilde{y}_{i+1,m}}{m} \tag{9}$$

In section 8.1 the simple model average is used to show how stacked forecast models are built in **mlr**. Section 8.2 does a more advanced method of ensemble averaging involving the forecast of endogeneous variables.

8.1 Stacking Univariate Learners

For this example, the models TBATS, GARCH, and ARFIMA [24] are stacked together and averaged on the climate task data. A resample description is made, and the function `makeLearners()` is used to start many learners at the same time.

```
resamp.sub = makeResampleDesc("GrowingCV",
                              horizon = 35L,
                              initial.window = .90,
                              size = nrow(getTaskData(climate.task)),
                              skip = .01
                              )
lrns = makeLearners(c("fcregr.tbats","fcregr.garch",
                      "fcregr.arfima"))
```

The function `makeStackedLearner()` takes the initialized learners and sets the meta information for stacking. This method uses simple model averaging

such as 9, however a super learner [46] can be used here, where $\phi()$ becomes another machine learning model.

```
stack.forecast = makeStackedLearner(base.learners = lrns,
                        predict.type = "response",
                        method = "average")
```

Each of the stacked learners are tuned over the cross product of all model parameters. This leads to a change in design where, given that some models may have the same argument names, the full name of the model is placed before the argument. Training over the cross-product of the model parameters leads to longer code, but it allows for a more honest perspective of how each model interacts in the stack.

```
# Simple param set for tuning sub learners
ps = makeParamSet(
  makeDiscreteParam("fcregr.tbats.h", values = 35),
  makeDiscreteParam("fcregr.garch.n.ahead", values = 35),
  makeDiscreteParam("fcregr.arfima.h", values = 35),
  makeDiscreteParam("fcregr.arfima.estim", values = "ls"),
  makeDiscreteParam(id = "fcregr.garch.model",
                    values = c("csGARCH")),
  makeIntegerVectorParam(id = "fcregr.garch.garchOrder",
                         len = 2L, lower = c(1),
                         upper = c(6)),
  makeIntegerVectorParam(id = "fcregr.garch.armaOrder",
                         len = 2L, lower = c(1),
                         upper = c(4)),
  makeDiscreteParam(id = "fcregr.garch.distribution.model",
                    values = c("norm","std","jsu")),
  makeDiscreteParam("fcregr.tbats.test",
                    values = c("kpss","adf","pp")),
  makeIntegerParam("fcregr.tbats.max.P", lower = 0, upper = 3),
```

```
  makeIntegerParam("fcregr.tbats.max.Q", lower = 0, upper = 2)
)
ctrl = makeTuneControlIrace(maxExperiments = 400L)
## tuning
library(parallelMap)
parallelStartSocket(7)
configureMlr(on.learner.error = "warn")
set.seed(1234)
fore.tune = tuneParams(stack.forecast, climate.task,
                       resampling = resamp.sub,
                       par.set = ps, control = ctrl,
                       measures = mase, show.info = FALSE)
parallelStop()
fore.tune
```

The rest of the modeling process flows in a way similar to the standard training and predicting schema. The function `setHyperPars2()` takes the best parameter models from the tuning process and assigns it to the final model to train over all of the data. Training and prediction operate in the same manner as univariate forecasters.

```
# get hyper params
stack.forecast.tune  = setHyperPars2(stack.forecast,fore.tune$x)
# Train the final best models and predict
stack.forecast.mod = train(stack.forecast.tune,climate.task)
stack.forecast.pred = predict(stack.forecast.mod,
                              newdata = m4.test)
stack.forecast.pred

## Prediction: 35 observations
## predict.type: response
## threshold:
## time: 0.00
##            truth response
## 2009-09-06     7 4.818892
```

```
## 2009-09-07    12 6.996835
## 2009-09-08    14 7.721747
## 2009-09-09    10 7.918065
## 2009-09-10    10 7.880770
## 2009-09-11    16 8.030163
## ... (35 rows, 2 cols)

performance(stack.forecast.pred,mase,climate.task)

##       mase
## 0.06057101
```

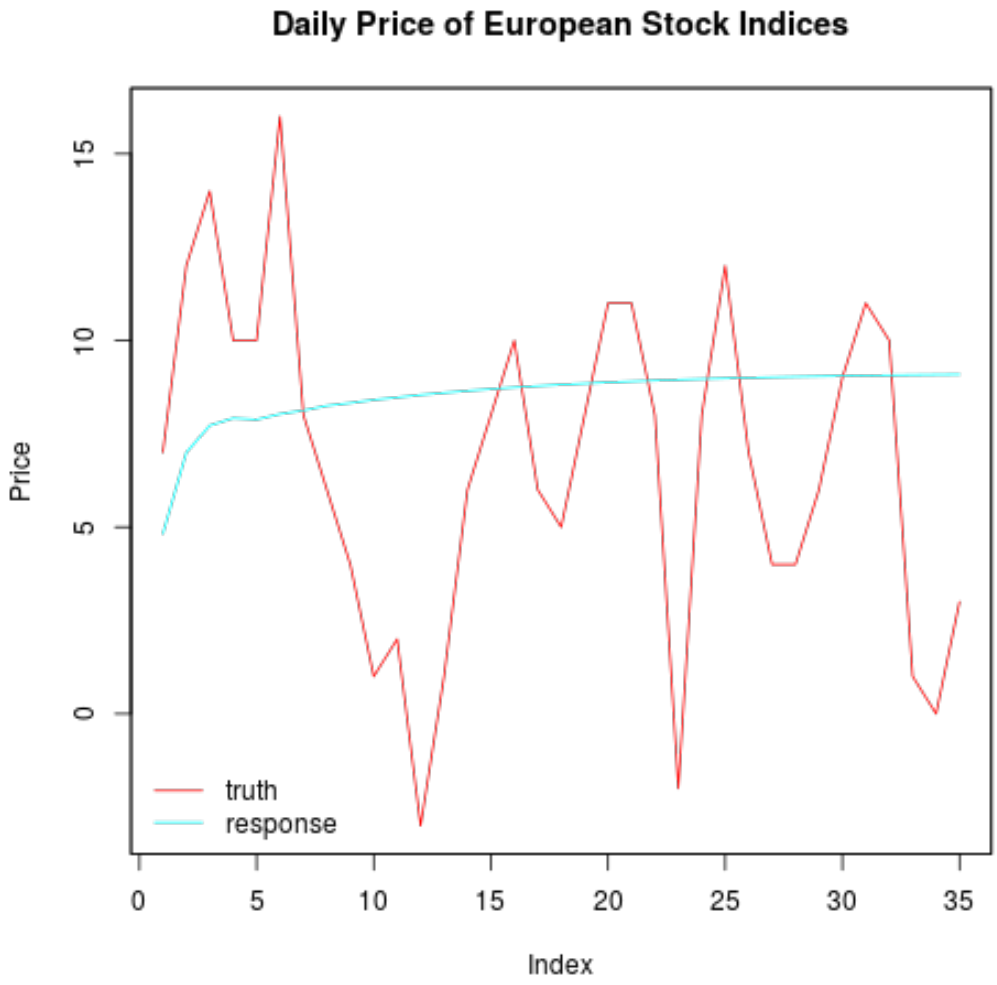


Figure 12: The forecast of the M4 data using the Stacked ARFIMA, TBATS, and GARCH model

8.2 Multivariate Stacked Learner

When there is a single target variable with multiple predictors stacked learning can be used with multivariate forecasters to forecast the predictors and have a machine learning model train over the forecasts of all variables. For this,

equation 8 can be modified to include forecasts of other predictors x_i, k where k is the index for each predictor variable

$$\tilde{y}_{i+1} = \phi(\tilde{y}_{i+1,1}, \ldots, \tilde{y}_{i+1,m}, \tilde{x}_{i+1,1}, \ldots, \tilde{x}_{i+1,k}, \sum_{j=1}^{m} \epsilon_{i+1,j}, \sum_{j=1}^{k} \epsilon_{i+1,j}) \tag{10}$$

In the example below, a boosted glm [8] is used as a super learner over a sparse lag multivariate VAR model to forecast FTSE prices. A resampling strategy is creating for both the underlying stacked learner and the super learner.

```
multfore.task = makeMultiForecastRegrTask(id = "bigvar",
                                          data = eu.train,
                                          target = "FTSE")

resamp.sub = makeResampleDesc("GrowingCV",
                              horizon = 32L,
                              initial.window = .90,
                              size = nrow(getTaskData(multfore.task)),
                              skip = .01
)

resamp.super = makeResampleDesc("CV", iters = 3)
```

In `makeStackedLearner()`, the super learner argument contains the boosted glm model.

```
base = c("mfcregr.BigVAR")
lrns = lapply(base, makeLearner)
lrns = lapply(lrns, setPredictType, "response")
lrns[[1]]$par.vals$verbose = FALSE
```

```
stack.forecast = makeStackedLearner(base.learners = lrns,
                     predict.type = "response",
                     super.learner = makeLearner("regr.glmboost",
                                                 family = "Laplace"),
                     method = "growing.cv",
                     resampling = resamp.sub)
```

Just as with univariate stacked forecasting models, a parameter set is created for the multivariate VAR model and tuning is done with `tuneParams()`.

```
ps = makeParamSet(
  makeDiscreteParam("mfcregr.BigVAR.p", values = 9),
  makeDiscreteParam("mfcregr.BigVAR.struct",
                     values = "SparseLag"),
  makeNumericVectorParam("mfcregr.BigVAR.gran", len = 2L,
                          lower = 35, upper = 50),
  makeDiscreteParam("mfcregr.BigVAR.h", values = 32),
  makeDiscreteParam("mfcregr.BigVAR.n.ahead", values = 32)
)

## tuning
library(parallelMap)
parallelStartSocket(4)
configureMlr(on.learner.error = "warn")
set.seed(1234)
multfore.tune = tuneParams(stack.forecast, multfore.task,
                     resampling = resamp.sub, par.set = ps,
                     control = makeTuneControlGrid(),
                     measures = mase, show.info = FALSE)
parallelStop()
```

Once the tuning is complete, `setHyperPar2()` extracts the final model parameters. Since the multivariate model is used to produce forecasts for a single variable, univariate MASE is used instead of the multivariate form of

MASE.

```
stack.forecast.f  = setHyperPars2(stack.forecast,
                                  multfore.tune$x)
multfore.train = train(stack.forecast.f,multfore.task)
```

```
multfore.pred = predict(multfore.train,
                  newdata = as.data.frame(eu.test))
multfore.pred

## Prediction: 32 observations
## predict.type: response
## threshold:
## time: 0.02
##                       truth response
## 1998-07-12 07:50:46 5960.2 5629.466
## 1998-07-13 17:32:18 5988.4 5629.378
## 1998-07-15 03:13:50 5990.3 5629.423
## 1998-07-16 12:55:23 6003.4 5629.519
## 1998-07-17 22:36:55 6009.6 5629.655
## 1998-07-19 08:18:27 5969.7 5629.759
## ... (32 rows, 2 cols)

performance(multfore.pred, mase, task = multfore.task)

##      mase
## 0.2363337
```

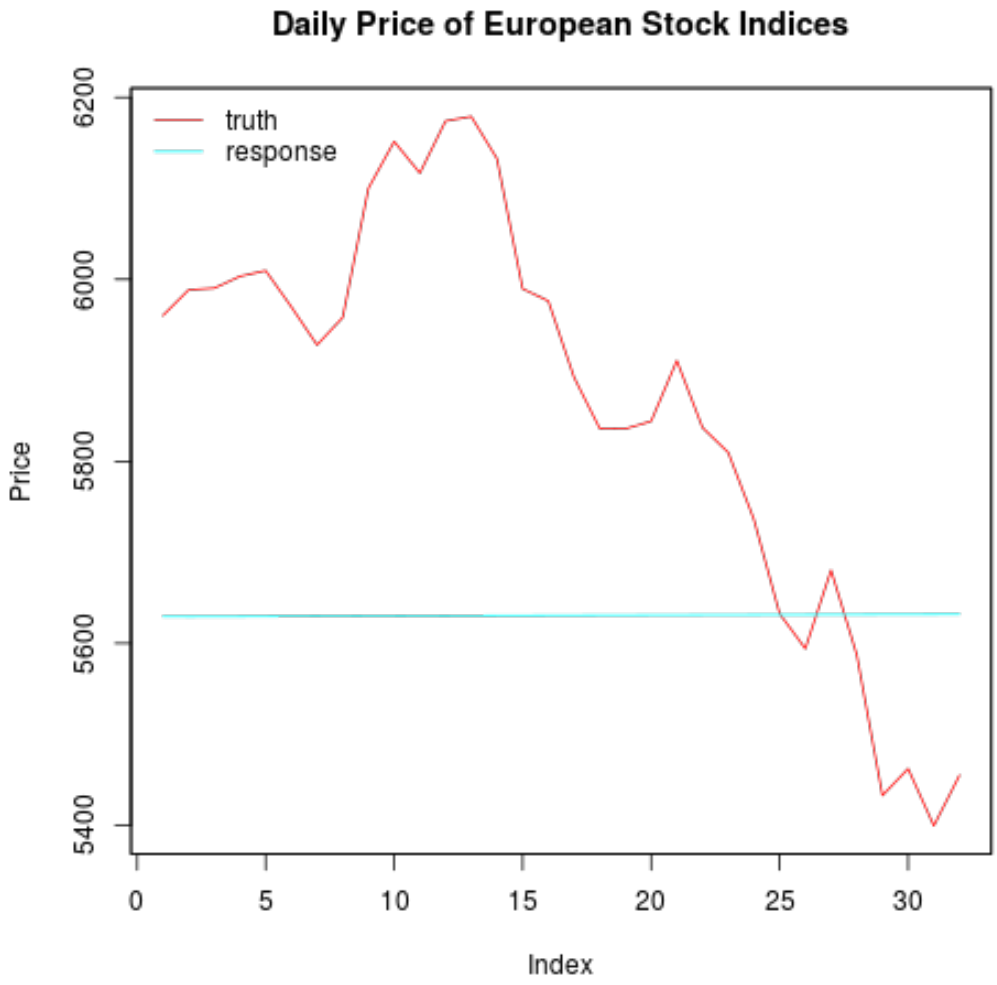


Figure 13: The forecast of the M4 data using the Stacked ARFIMA, TBATS, and GARCH model

This model only predicts a mean value for all forecasts, which may be due to poor performance of the stacked learner on top of the forecasts. The super learner trains on the forecasts of all the windowed cross-validations, though it does not understand that the forecast in period one is going to be better than the forecast at period n. Future research will create methods to correct this

bias while training the super learner such as regressive weights for over the forecast period, giving more weight to the most recent forecast.

9 Conclusion

The results of this paper show that creating a unified interface for forecasting models in R allows for better models through an automated methodology of resampling, preprocessing, model selection, stacking tuning, and training. Building on the broad range of forecasting packages available in R, automating tasks such as windowing cross-validation and model selection allow applied forecasters to spend less time dealing with the bureaucracy of modeling and more time testing new models. New methods such as multivariate stacked learners, Lambert W transforms, and the ability to create arbitrary AR(p, d) machine learning models allows researchers to experiment with new ideas easily. While the example models here are not perfect, this was mostly due to time. It will be easy for researchers to beat the models created in this paper.

Future research based on this package would involve tuning these models to see how useful they are in the real world. The classification forecast learners made available in the forecasting extension of **mlr** is a growing field, and with the ease of making models in **mlr** new research in this area can progress be easily replicable. Updates to this package will include more multivariate and univariate forecast learners as well as new methods to stack models such as Bayesian averaging [41].

References

[1] *CMA-ES: A Function Value Free Second Order Optimization Method*, Paris, France, 2014.

[2] Robert L. Winkler Allan H. Murphy. Probability forecasting in meterology. *Journal of the American Statistical Association*, 79(387):489–500, 1984.

[3] Souhaib BenTaieb. *M4comp: Data from the M4 Time Series Forecasting Competition*, 2016. R package version 0.0.1.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305, February 2012.

[5] Bernd Bischl and Michel Lang. *parallelMap: Unified Interface to Parallelization Back-Ends*, 2015. R package version 1.3.

[6] Bernd Bischl, Michel Lang, Jakob Bossek, Daniel Horn, Jakob Richter, and Pascal Kerschke. *ParamHelpers: Helpers for Parameters in Black-Box Optimization, Tuning and Machine Learning*, 2016. R package version 1.9.

[7] Bernd Bischl, Michel Lang, Jakob Richter, Jakob Bossek, Leonard Judt, Tobias Kuehn, Erich Studerus, and Lars Kotthoff. *mlr: Machine Learning in R*, 2015. R package version 2.7.

[8] Peter Buehlmann and Torsten Hothorn. Boosting algorithms: Regularization, prediction and model fitting (with discussion). *Statistical Science*, 22(4):477–505, 2007.

[9] Thomas Chadefaux. Early warning signals for war in the news. *Journal of Peace Research*, 51(1):5–18, 2014.

[10] Jiangpeng Chen, Xun Lei, Li Zhang, and Bin Peng. Using extreme value theory approaches to forecast the probability of outbreak of highly pathogenic influenza in zhejiang, china. In *PloS one*, 2015.

[11] Robert T. Clemen. Combining forecasts: A review and annotated bibliography. *International Journal of Forecasting*, 5(4):559 – 583, 1989.

[12] Thomas G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00, pages 1–15, London, UK, UK, 2000. Springer-Verlag.

[13] J. Rissanen E. J. Hannan. Recursive estimation of mixed autoregressive-moving average order. *Biometrika*, 69(1):81–94, 1982.

[14] Graham Elliott and Robert P. Lieli. Predicting binary outcomes. *Journal of Econometrics*, 174(1):15 – 26, 2013.

[15] Robert F. Engle. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, 50(4):987–1007, 1982.

[16] Philip Hans Franses. A note on the Mean Absolute Scaled Error. *International Journal of Forecasting*, 32(1):20–22, 2016.

[17] Jerome H. Friedman. On bias, variance, 0/1—loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery*, 1(1):55–77, 1997.

[18] Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer, Brenton Kenkel, the R Core Team, Michael Benesty, Reynald Lescarbeau, Andrew Ziem, Luca Scrucca, Yuan Tang, and Can Candan. *caret: Classification and Regression Training*, 2015. R package version 6.0-62.

[19] Alexios Ghalanos. *rugarch: Univariate GARCH models.*, 2015. R package version 1.3-6.

[20] Georg M. Goerg. Lambert w random variables - a new family of generalized skewed distributions with applications to risk estimation. *Annals of Applied Statistics*, 5(3):2197–2230, 2011.

[21] Georg M. Goerg. The lambert way to gaussianize heavy-tailed data with the inverse of tukey's h transformation as a special case. *The Scientific World Journal: Special Issue on Probability and Statistics with Applications in Finance and Economics*, 2015(2015):16, 2015.

[22] Alex Goldstein, Adam Kapelner, Justin Bleich, and Emil Pitkin. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics*, 24(1):44–65, 2015.

[23] Jan G. De Gooijer and Rob J. Hyndman. 25 years of time series forecasting. *International Journal of Forecasting*, 22(3):443 – 473, 2006. Twenty five years of forecasting.

[24] C. W. J. Granger and Roselyne Joyeux. An introduction to long-memory time series models and fractional differencing. *Journal of Time Series Analysis*, 1(1):15–29, 1980.

[25] Clive W.J. Granger. Forecasting stock market prices: Lessons for forecasters. *International Journal of Forecasting*, 8(1):3 – 13, 1992.

[26] David C. Hoaglin. *Summarizing Shape Numerically: The g-and-h Distributions*, pages 461–513. John Wiley & Sons, Inc., 2006.

[27] R.J. Hyndman and G. Athanasopoulos. *Forecasting: principles and practice:*. OTexts, 2014.

[28] Rob J Hyndman and Yeasmin Khandakar. Automatic time series forecasting: the forecast package for R. *Journal of Statistical Software*, 26(3):1–22, 2008.

[29] Rob J. Hyndman and Anne B. Koehler. Another Look at Measures of Forecast Accuracy. Monash Econometrics and Business Statistics Working Papers 13/05, Monash University, Department of Econometrics and Business Statistics, May 2005.

[30] Max Kuhn. caret data splitting methods, 1999.

[31] Kajal Lahiri and Liu Yang. *Forecasting Binary Outcomes*, volume 2 of *Handbook of Economic Forecasting*, chapter 0, pages 1025–1106. Elsevier, May/June 2013.

[32] Mark T. Leung, Hazem Daouk, and An-Sing Chen. Forecasting stock indices: a comparison of classification and level estimation models. *International Journal of Forecasting*, 16(2):173 – 190, 2000.

[33] Mark T. Leung, Hazem Daouk, and An Sing Chen. Forecasting stock indices: A comparison of classification and level estimation models. *International Journal of Forecasting*, 16(2):173–190, 4 2000.

[34] Alysha M. De Livera, Rob J. Hyndman, and Ralph D. Snyder. Forecasting time series with complex seasonal patterns using exponential smoothing. *Journal of the American Statistical Association*, 106(496):1513–1527, 2011.

[35] Spyros Makridakis and Michèle Hibon. The m3-competition: results, conclusions and implications. *International Journal of Forecasting*, 16(4):451 – 476, 2000. The M3- Competition.

[36] Stephan Morgenthaler and John W. Tukey. Fitting quantiles: Doubling, hr, hq, and hhh distributions. *Journal of Computational and Graphical Statistics*, 9(1):180–195, 2000.

[37] W. Nicholson, D. Matteson, and J. Bien. VARX-L: Structured Regularization for Large Vector Autoregressions with Exogenous Variables. *ArXiv e-prints*, August 2015.

[38] Will Nicholson, David Matteson, and Jacob Bien. *BigVAR: Dimension Reduction Methods for Multivariate Time Series*, 2016. R package version 1.0.1.

[39] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[40] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016. The data were kindly provided by Erste Bank AG, Vienna, Austria.

[41] Adrian E. Raftery, Tilmann Gneiting, Fadoua Balabdaoui, and Michael Polakowski. Using bayesian model averaging to calibrate forecast ensembles. *Monthly Weather Review*, 133(5):1155–1174, 2005.

[42] David Reilly. The autobox system. *International Journal of Forecasting*, 16(4):531–533, 2000.

[43] Goodrich RL. The forecast pro methodology. *International Journal of Forecasting*, 16(4):533–535, 2000.

[44] J. D. Rodriguez, A. Perez, and J. A. Lozano. Sensitivity analysis of k-fold cross validation in prediction error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(3):569–575, March 2010.

[45] Jeffrey A. Ryan and Joshua M. Ulrich. *xts: eXtensible Time Series*, 2016. R package version 0.10-0.

[46] David H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[47] Tuncel M. Yegulalp. Forecasting for largest earthquakes. *Management Science*, 21(4):418–421, 1974.