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

Steve Bronder

sab2287@columbia.edu

December 22, 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**. It includes standard and novel tools such as autoregressive and LambertW transform data generating processes, fixed and growing window cross validation, and forecasting models in the context of univariate and multivariate time series. Examples from forecasting competitions will be given in order to demonstrate the benefits of a unified framework for machine learning and time series.

1 Introduction

There has been a rapid developement in time series methods over the last 25 years [20] whereby time series models have not only become more common, but more complex. The R language [36] has several task views with compiling the packages available for forecasting, time series methods, and applied finance. However, the open source nature of R has left users without a standard framework. Many packages have their own sub-culture of style, syntax, and output. The **mlr** [5] package, short for Machine Learning in R, works to give a strong syntatic framework for the modeling pipeline. By automating many of the standard tools in machine learning such as preprocessing and cross validation, **mlr** reduces error from the user during the modeling process.

While there are some time series methods available in **caret** [16], 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 choice for implementing time series methods and models. This paper will show how using **mlr**'s strong syntatic structure allows for time series packages such as **forecast** [25], **rugarch** [17], and **BigVAR** [35] to use machine learning methedologies such as automated parameter tuning, data preprocessing, model blending, cross validation, performance evaluation, and parallel processing techniques for decreasing model build time.

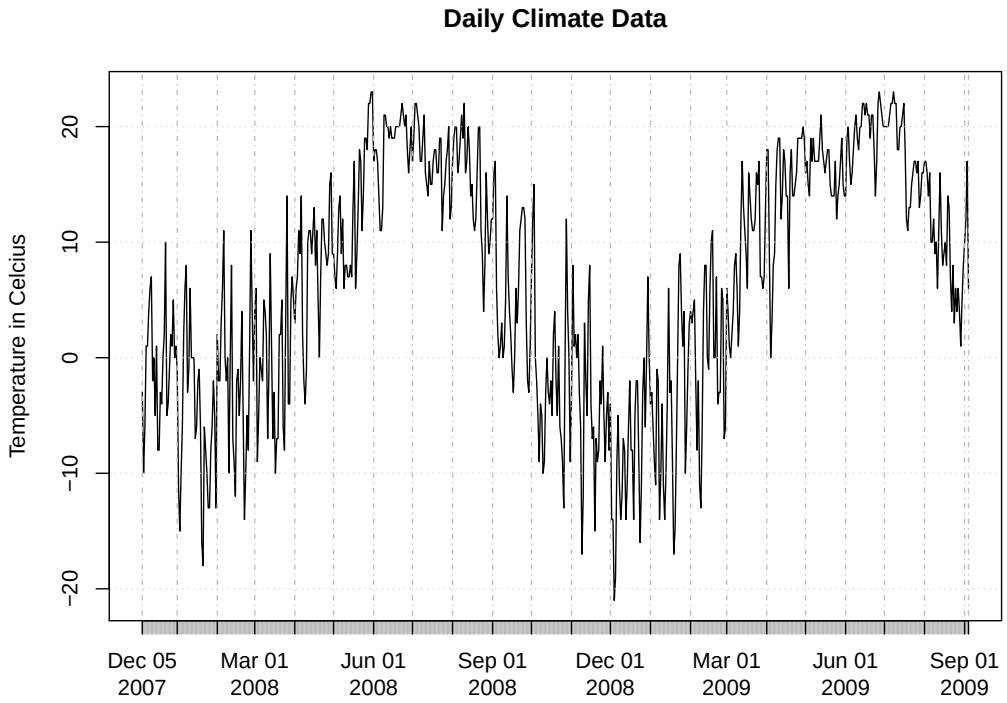
2 Forecasting Example with the M4 Competition

Professional forecasters attempt to predict the future of a series based on its past values. Forecasting can be used in a wide range of tasks including forecasting stock prices, [22], weather patterns [1], international conficts [7], and earthquakes [44]. In order to evaluate **mlr**'s forecasting framework we need a large set of possible time series to make sure our methods generalize well.

The Makridakis competition [32] 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. To show examples of how **mlr**'s forecasting features works we will look at a particular climate series. 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.

```
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.cli
```

```
m4.test <- xts(m4.climate$future, as.POSIXct("2009-09-06") + days(0:I(length(m4.cl
colnames(m4.train) <- "target_var"
colnames(m4.test) <- "target_var"
```

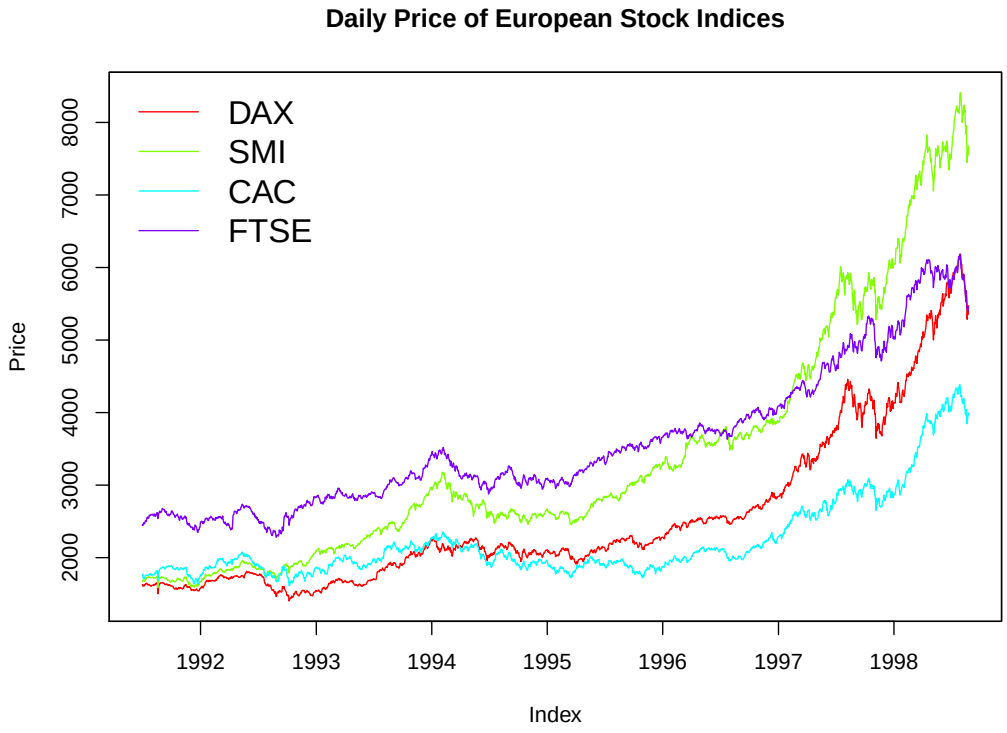


This series was chosen for its obvious seasonality and time features. Some series in M4 only contain 12 observations, which is not enough data to accurately train a model. The example data set should be large enough that the tuning method can take multiple windows of the data. We can see figure one is what most people imagine when they think of a time series. There is a clear seasonal time trend with individual points moving about the seasonal periods. The data can be found in the package **M4comp** [2] under sets `M4[28]` and `M4[29]`.

For multivariate forecasting, we will use the EUStockMarkets data set from the **datasets** [37]. It contains a set of DAX, SMI, CAC, and FTSE European

stock indices from July 1st, 1991 to August 24th, 1998 totaling 1828 training observations and 32 test observations.

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



Note that each stock index tends to follow a similar, but diverging, trend. This will be important to note when we perform windowing cross validation as it will let us see how well the models adapt to what appears to be nonstationary data.

3 Univariate and Multivariate Forecasting Tasks

4 Univariate Tasks

mlr uses the S3 object system to clearly define a predictive modeling task. Tasks contain the data and other relevant information such as the task id and which variable you are targeting for supervised learning problems. Forecasting tasks are handled in **mlr** by the function `makeForecastRegrTask()`. The forecasting task inherets most of it's arguments from `makeRegrTask`, but has two noticable differences in arguments.

data: Instead of a data frame, an xts object from **xts** [42] containing the time series.

frequency: An integer with 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 we have missing values. Note that there are zero features in our task because we only have a target variable, which the model itself will use to build features.

4.1 Multivariate Tasks

One common problem with forecasting is that it is difficult to use additional explanatory variables or forecast multiple targets that are dependent on one another. If we are at time t and want to forecast 10 periods in the future, we need to know the values of the explanatory variables at time $t + 10$, which is often not possible. A new set of models [35] which treats explanatory variables endogenously instead of exogenously allows us to forecast not only our target, but addititional explanatory variables. This is done by treating all the variables as targets, making them endogeneous to the model. To use these models, we create a multivariate forecasting task. 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 endogeneously.

```
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 the output, predictions, and even using the measures for univariate forecasting tasks. Both of these tasks will still forecast all of the underlying series, which allows us take exogeneous models and treat them endogeneously for n-step 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
```

5 Building and Tuning a forecast learner

5.1 Univariate Forecasting

The `makeLearner()` function provides a structured model building framework to the several forecasting models currently implimented in **mlr**. As an example, we will build the Trigonometric exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal Components (TBATS) [31].

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

and autoregressive neural networks. In addition, preprocessing features have been added to allow arbitrary supervised machine learning models to be used in the context of forecasting. To impliment the TBATS model we use `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")
```

We can also supply a predict type for forecasting models to either receive point estimates (`response`) or point estimates with quantiles of confidence intervals (`quantile`). To train the model we simply call train, supplying the forecasting model and task. After training the model it's simple to get our forecasts by calling `predict()` with the test data, returning an object containing meta information for the forecasts along with the prediction and test data in columns `truth` and `response`, respectively.

```
train.tbats= train(learner = tbats.mod, task = climate.task )
predict.tbats = predict(train.tbats, newdata = m4.test)
```

To measure the performane of TBATS we call `performance()` with the Mean Absolute Scaled Error (MASE) [26] measure.

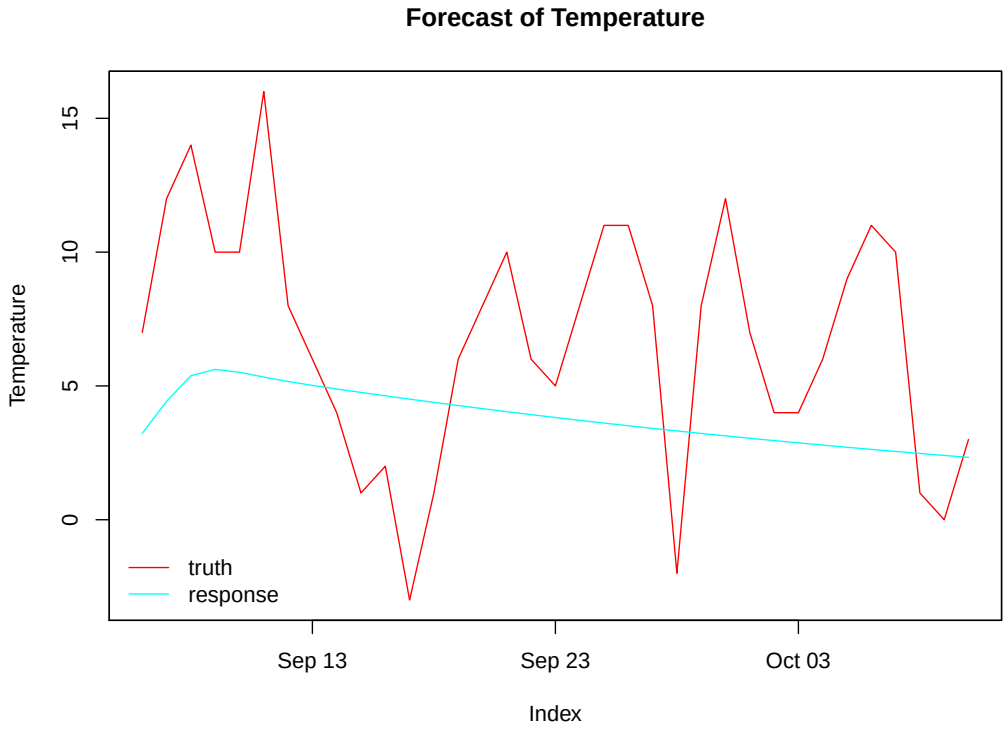
```
performance(predict.tbats, mase, task = climate.task)

##      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's very interpretable. Let y_t and $\tilde{y_t}$ be the target variable and prediction at time t until the final time T with $\epsilon_t = y_t - \tilde{y_t}$ being the forecast error.

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

Where the denominator is the one step ahead naive forecast from the training data. When the numerator is equal to the denominator the model performed as good as a simple naive forecast method. Scores greater than one mean you are performing worse and scores less than one mean you are performing better than the naive forecasting method.

The scale invariance of MASE means that it is independent of the scale of the data which allows models to be compared across data sets. The scale invariance of MASE has made it a favorite for comparing the accuracy of forecast methods [14] 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. This allows MASE to be use in situations in which zeros occur frequently or zero is not meaningful such as predicting temperature.



Need to talk about results

Because the forecasts are only for the next 35 periods it's useful to be able to update the model continuously without retraining the model each time we want new forecasts. Univariate forecasting models in **mlr** can be updated using `updateModel()`.

```
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)
```

5.2 Multivariate Forecasting

The package **BigVAR** has been implemented for multivariate forecasting. **BigVAR** allows for estimation of high dimensional time series through including structured Lasso penalties to the vector autoregression framework [34].

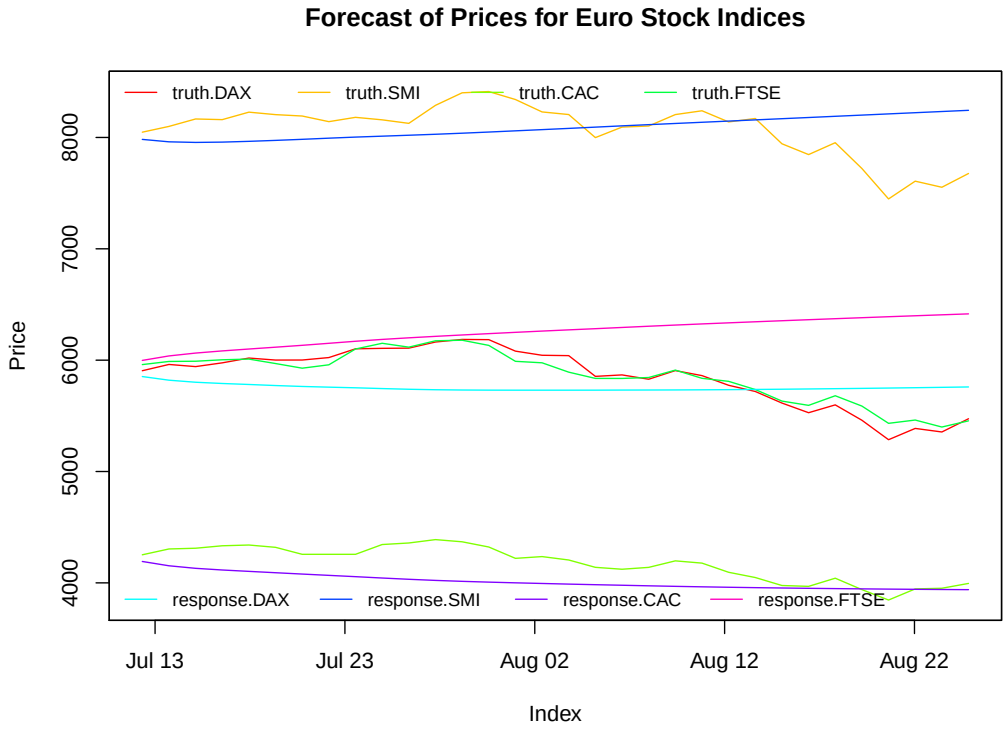
```
bigvar.mod = makeLearner("mfcregr.BigVAR",p = 25, struct = "SparseLag",
                         gran = c(50, 60),h = 35, n.ahead = 35)
```

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

```
## 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. A multivariate version of MASE has been implimented which takes the mean of each MASE score for the individual variables.

```
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
```



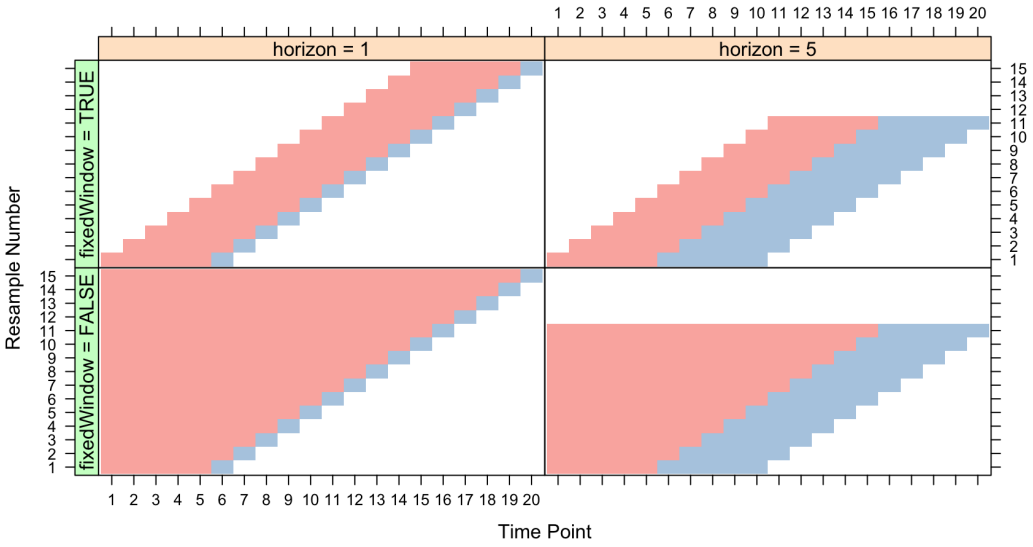
6 Resampling with Time

While TBATS is one of the most well known time series models, the order selection process or 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 [11] 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 regresssions. Proprietary algorithms from software such as Forecast

Pro [40] and Autobox [39] are well known and have performed to high standards in competitions such as the M3 forecasting competition [32]. One of the most well known R packages for automated forecast is **forecast** [25] which contains several methods for automated forecasting including exponential smoothing based methods and step-wise algorithms for forecasting with 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. To do this, we will use a windowing resampling scheme to train over the possible models. Resampling schemes such as cross-validation, bootstrapping, etc. are common in machine learning for dealing with the bias-variance tradeoff [15] [41]. When their is a time component to the data, windowing schemes are useful in allowing a valid resampling scheme while still maintaining the time properties of the series. Figure one gives an example of fixed and growing windows. Given a horizon and initial starting point the window slides forward one step each time while either shifting in the fixed case or enlarging by one in the growing case. Growing and fixed window resampling such as from [24] are now available in the `resampling()` function of **mlr**.

Figure 1: Resampling with a window scheme as exampled by caret [27]. The top graphs are fixed window cross validation while the bottom graphs are growing window cross validation.



A windowing resampling process is created in the function `makeResampleDesc()` by supplying the resampling type, 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
```

To make a parameter set to tune over **mlr** uses **ParamHelpers** [4]. There are several types of tools to help us search our parameter space including grid search, random search [3], to search our 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`

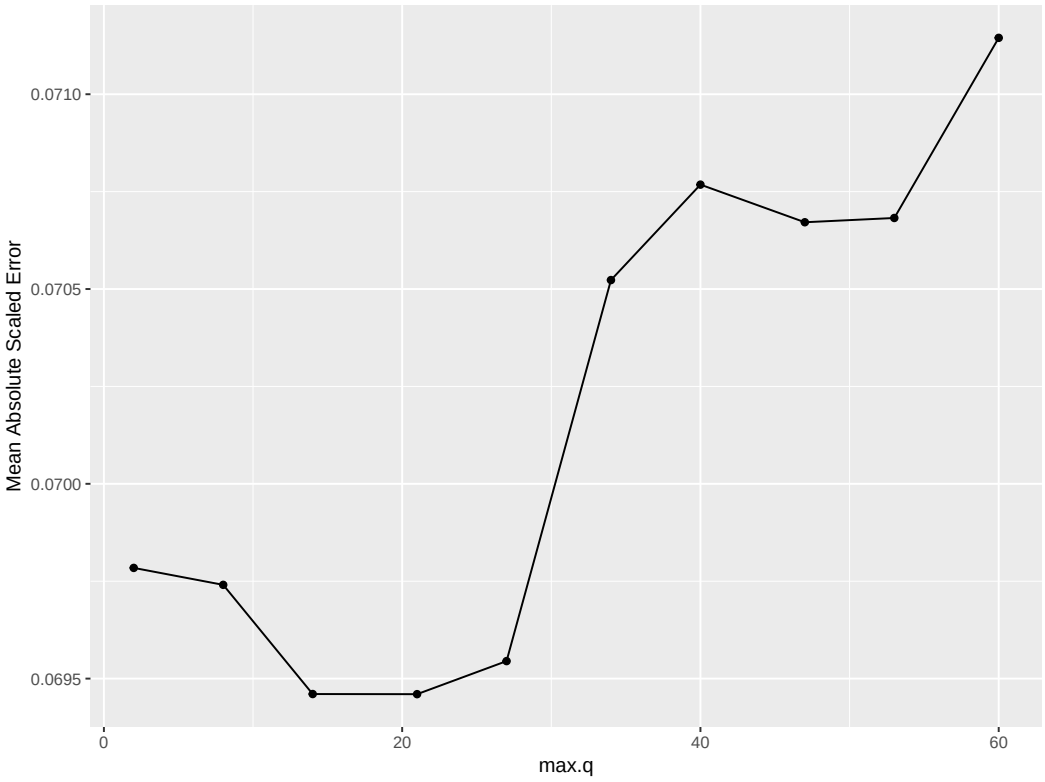
argument allows the tuning process to take values on a seperate scale than the one described. In the tuning scheme above, trafo will look search over $1, 4, 9, ..., 81$, and 100 values of lag, selecting the model with the best lag structure. For this tuning task we use MASE [26] as a measure of performance [1].

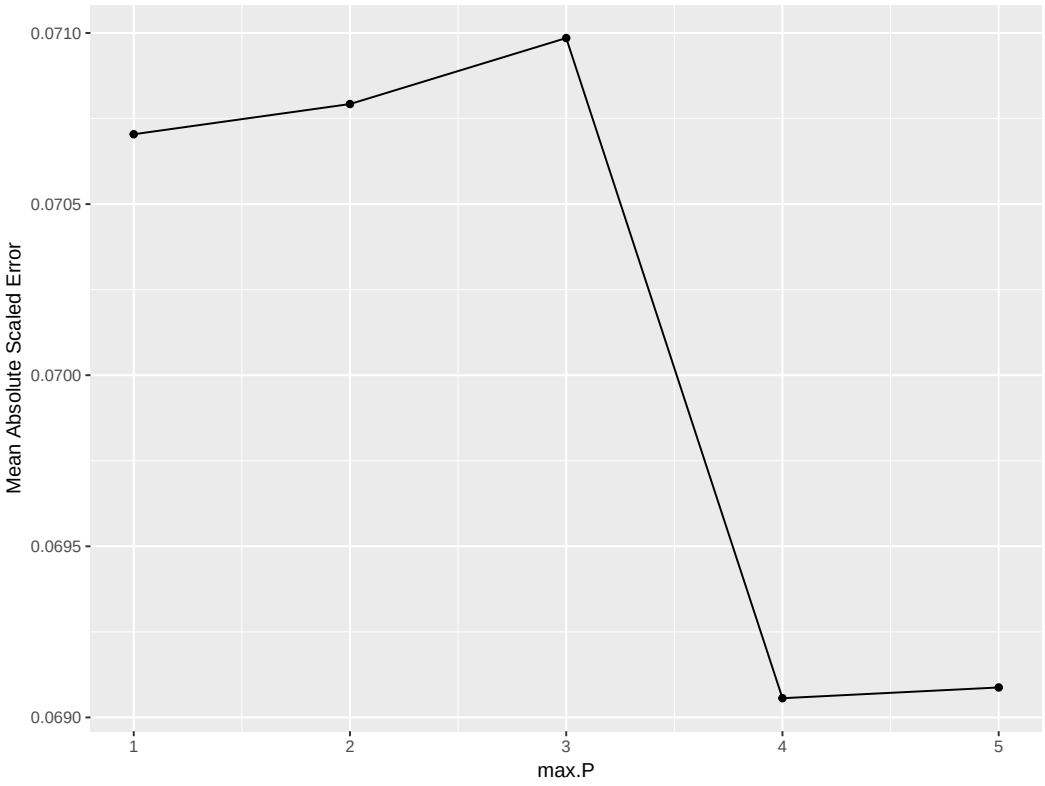
```
#
library("parallelMap")
parallelStartSocket(8)
configureMlr(on.learner.error = "warn")
set.seed(1234)
tbatsTune = tuneParams(makeLearner("fcregr.tbats", h = 35),
                       task = climate.task,
                       resampling = resampDesc, par.set = parSet,
                       control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

[1] Models with a seasonal difference > 0 may be favorably biased as we use the non-seasonal MASE score


0.0710
0.0705
0.0700
0.0695
Mean Absolute Scaled Error
0
20
40
60
max.q



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.

```
lrn = setHyperPars(makeLearner("fcregr.tbats", h = 35),
                   par.vals = tbatsTune$x)
m = train(lrn, climate.task)
```

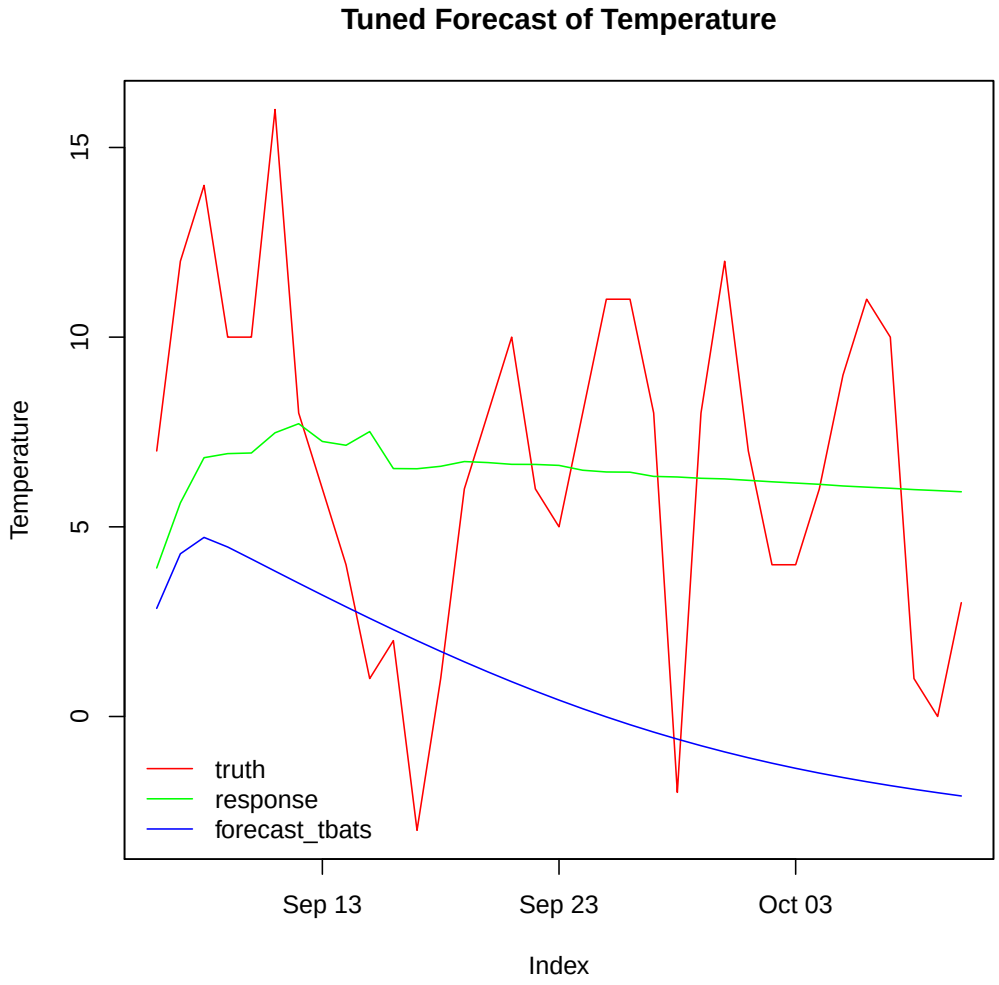
To make predictions for our test set we simply pass our model, task, and test data to `predict()`

```
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's possible to see the clear benefit of the forecast extension to **mlr**. While **forecast** does some automated tuning, this model would still require extensive manual testing in an attempt to find the best model. By automating this process we come to a faster solution with minimal work.



To this author's knowledge, this is the first package in R that allows for automated tuning and training of GARCH models [13]. It is possible to pass and train multiple types of GARCH models while also tuning the models respective parameters. In this example we also use `predict.type = "quantile"` to estimate confidence intervals for the forecast.

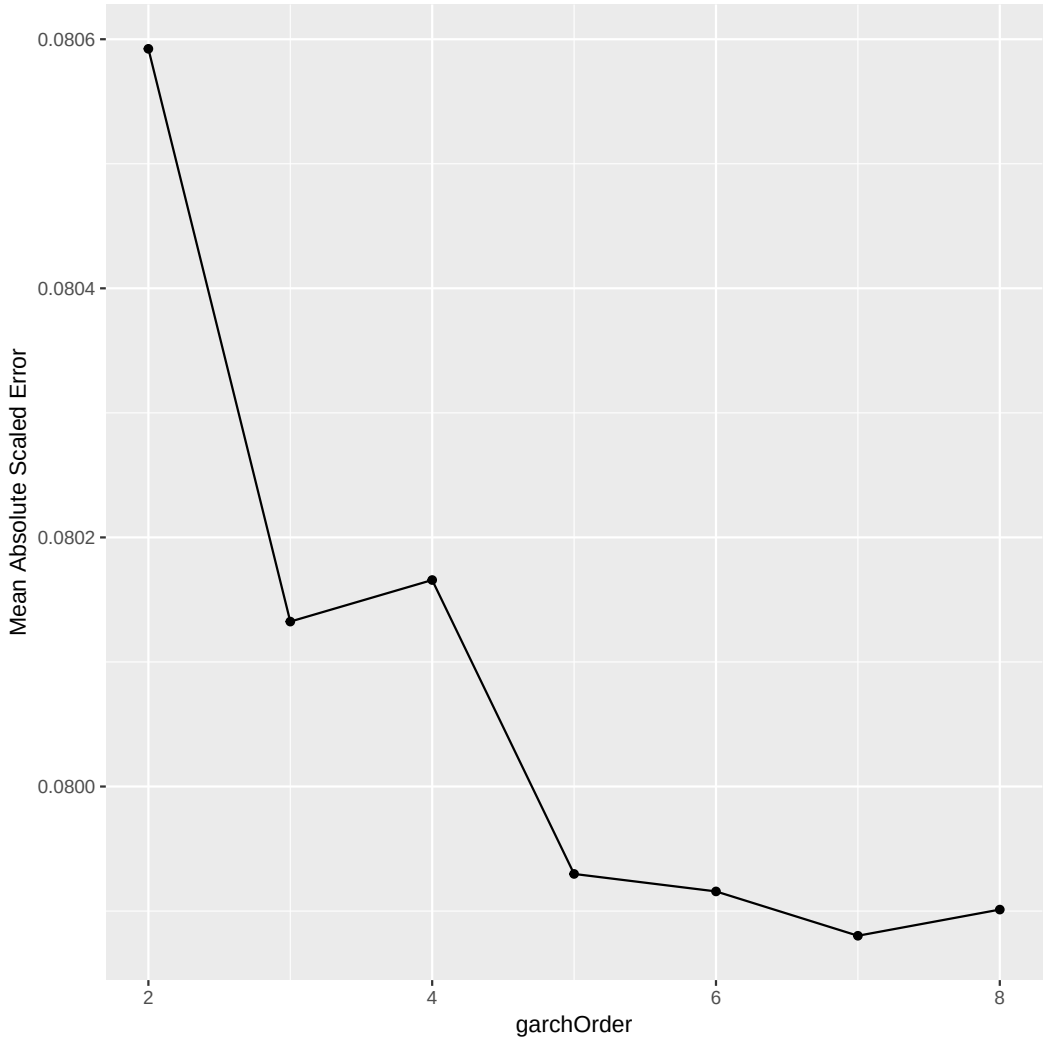
```
par_set = makeParamSet(
  makeDiscreteParam(id = "model", values = c("sGARCH", "csGARCH")),
  makeIntegerVectorParam(id = "garchOrder", len = 2L, lower = c(1,1),
                         upper = c(4,4)),
  makeIntegerVectorParam(id = "armaOrder", len = 2L, lower = c(5,1),
                         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 grid estimation
ctrl = makeTuneControlIrace(maxExperiments = 400L)

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
```

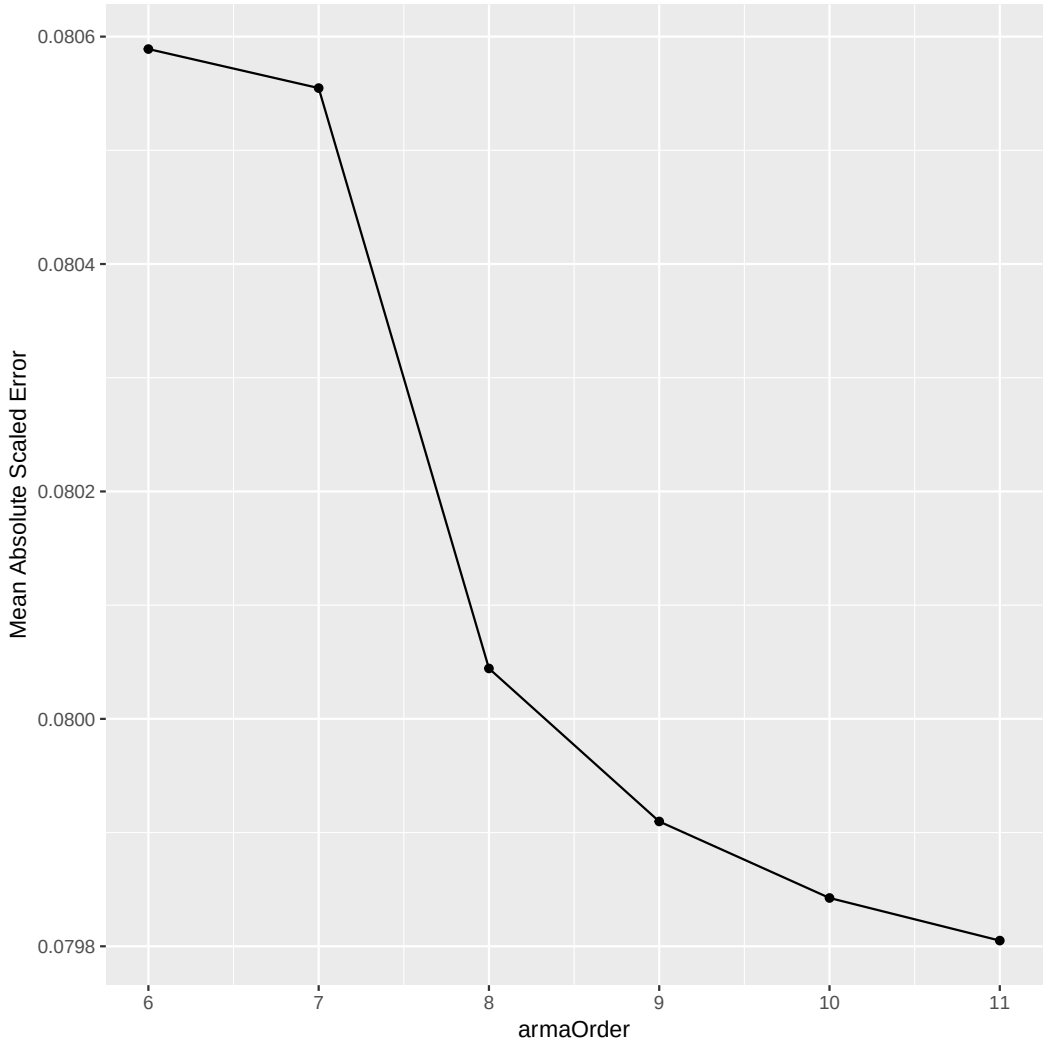
Because the partial dependence plots are still in development they do not work well with `makeNumericVectorParams()` such as `garchOrder` and `armaOrder`. Instead of looking at each p, q individually, we simply sum them and look at the maximum coefficient values for the p, q order of the GARCH

component and the ARMA component.

```
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")
```



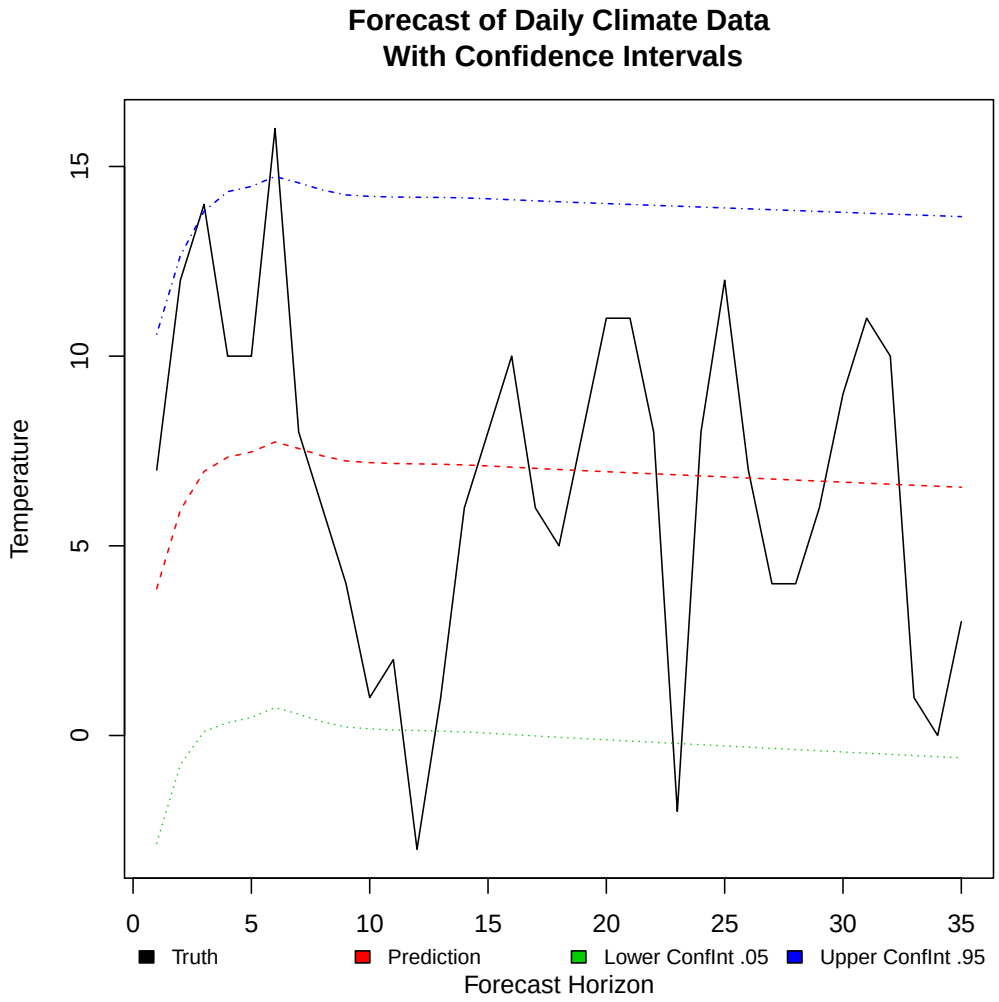

```
plotHyperParsEffect(garch.hyperpar, x= "armaOrder", y = "mase.test.mean",
                    plot.type = "line",
                    partial.dep.learn = "regr.randomForest")
```



The dependence plots show that for both the GARCH and ARMA orders, as the order is increased, the MASE score decreases. We would most likely find a better model if we allowed the order of the model to increase up to around fifteen or twenty or both. Forecasting models with quantiles are treated the same as any other learner post-tuning. The best hyperparameters are taken and a final model is trained over the entire dataset.

```
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)

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 heteroskedasticity of the past series, the TBATS model performs well in terms of catching long or multiseasonal type periods. Section 9.1 goes over how to combine forecasting models to make

benefit of both models of their strengths.

7 Forecasting with Machine Learning Models

7.1 Forecasting with Regression Tasks

The forecasting extension of **mlr** includes a preprocessing function that allows 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.

```
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

$$\hat{y}_{t+1} = \sum_{i=1}^{p} (\rho_i \Delta_d y_i + \epsilon_i) \tag{2}$$

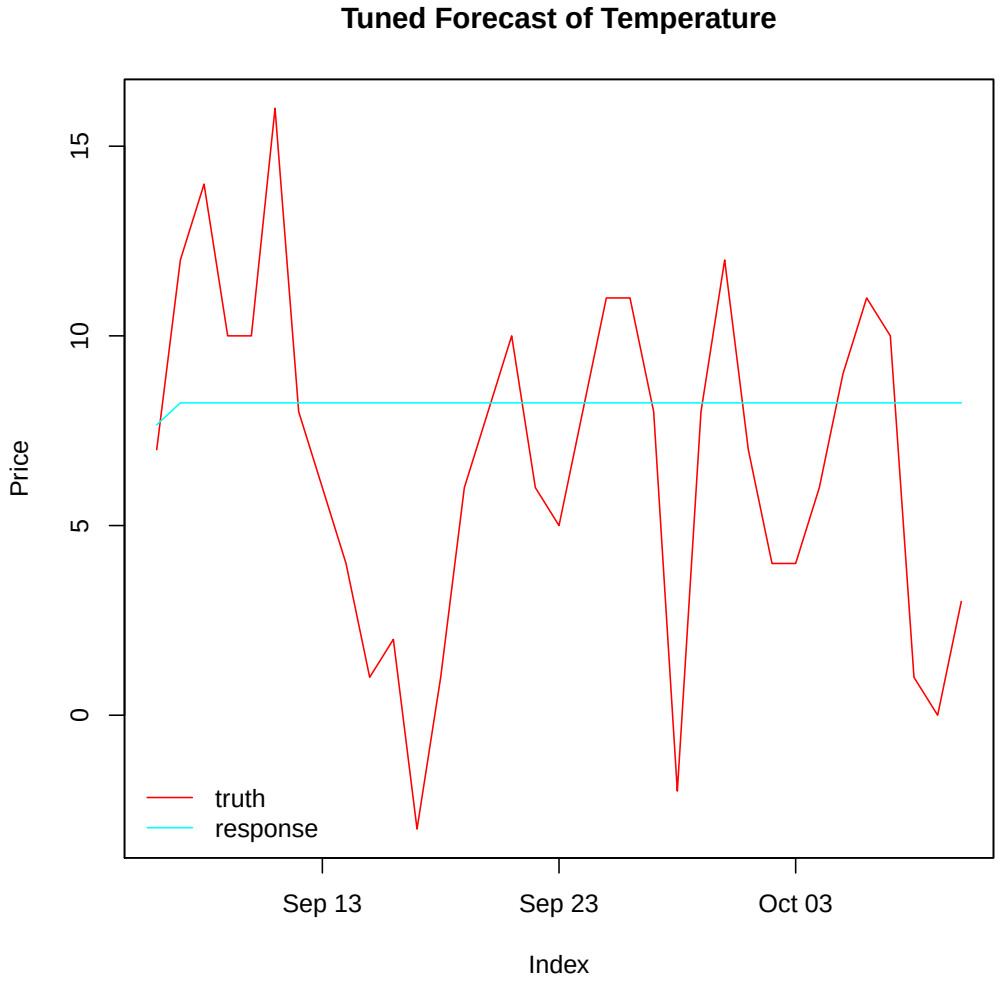
where ρ_i is the autoregressiver parameter of order p and d is the lag of the difference operator Δ. Then the n-step ahead forecast is defined as

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

```
gbm.forecast = forecast(gbm.train, h = 35L,
                        newdata = as.data.frame(m4.test))
performance(gbm.forecast,mase,climate.task.lag)
```

```
## 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)
##      mase
## 0.0558791
```



7.2 Forecasting with Classification Tasks

Forecasting for binary or multiclass outcomes [28] is a common problem in the real world. However, research in this area of forecasting only started picking up speed in the last decade [12]. With a this new extension to **mlr** researchers

now have the ability to take all the classification models in **mlr** and apply them to the forecasting context. For developing trading strategies, we normally have a discrete set of choices such as to buy, sell, or hold onto a stock. Using forecasting in mlr we can now train classification models that forecast these choices [29]. To example this, a simple buy, sell, or hold trading strategy will be built Using the `EuStockMarkets`'s DAX index. If the stock goes up by 5% in a day we will buy, down 5% we will sell, and otherwise we will hold onto the current stocks we have.

```
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))
# Note: Error on multiples of the same observation
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")
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)
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 methedologies available in **mlr**, forecasting binary outcomes is now as simple as any other model. These tools can be used for further research in areas such as directions of stock movement [30] and forecasting extreme values [8].

8 Lambert W Transforms

Many machine learning and time series models rely one the assumption that our 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 Transforms are a family of generalized skewed distributions [18] that have bijecetive and parametric functions that allow heavy tailed and asymettric data to appear more Gaussian [19].

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

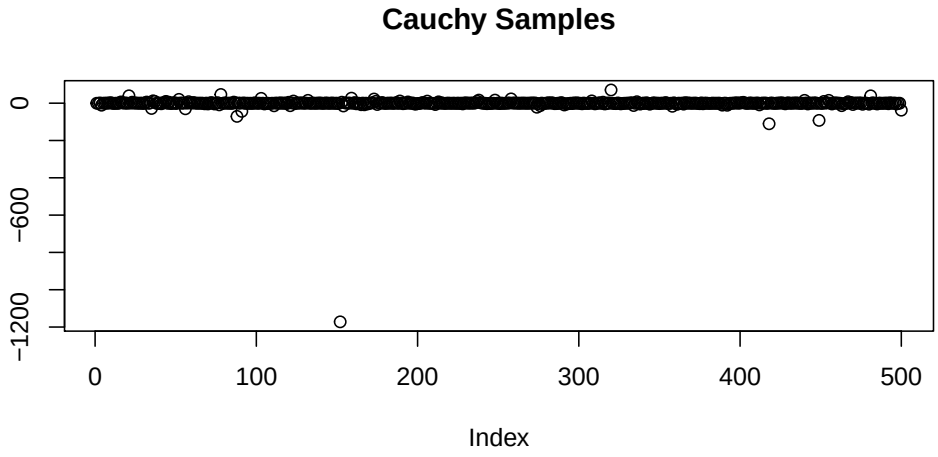
$$test \tag{4}$$

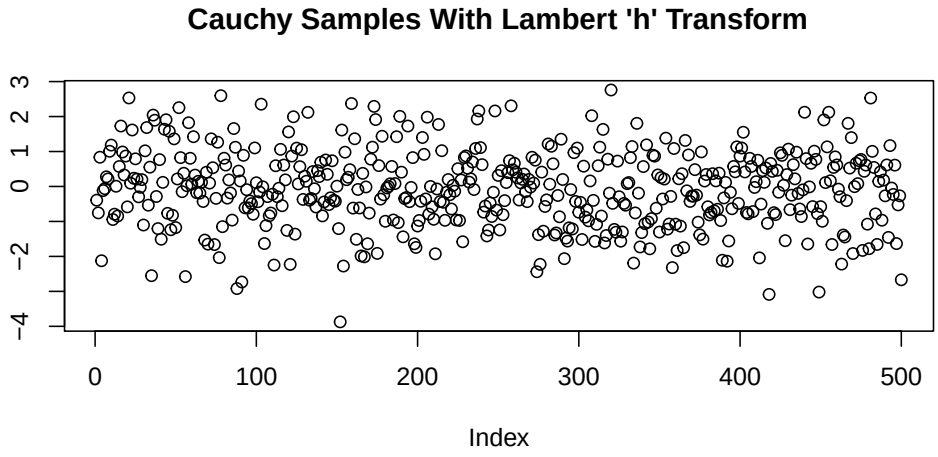
And the heavy-tailed Lambert $W \times F_X$ random variable can be defined as

$$test \tag{5}$$

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 12 reduces to a standard normal distribution. Equation 12 is the general form of Tukey's h distribution [23] and the basis for Morgenthaler and Tukey's [33] skewed, heavy tailed family of hh random variables.

$$Z = \begin{cases} U \exp\left(\frac{\delta_l}{2}(U^2)_l^{\alpha}\right), \ \delta_l \geq 0 \ \alpha_l > 0 \\ U \exp\left(\frac{\delta_r}{2}(U^2)_r^{\alpha}\right), \ \delta_r \geq 0 \alpha_r > 0 \end{cases} \tag{6}$$





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. Wrapping the model in this way allows the pre-processing function to be intertwined with the model itself. There are opportunities for users to accidentally bias there own results due to improperly applying pre-processing schemes. 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 seperatly. But than what about cross-validation? The goal of cross-validation follows the above schema as well and it is very easy for a user, who assumes they have made a good faith attempt to not bias their model, will end up with overconfident results. Making the preprocessing part of the model itself allows **mlr** to overcome this.

There will be a step for the training data or subset, for example to demean it, and then there will be a function for prediction on the testing data, or in the example above to demean the test set alone. In the context of Lambert W$\times$ F() transforms, the h, hh, or s distribution that gaussianizes the data is estimated 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 methedology, creating the model with Lambert preprocessing, training the model, and then performing prediction. The end result to the user appears the same, but a significant amount of bias is reduced in the background.

```
# Need to make this more dramatic
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
```

9 Stacking Forecasting Learners

Stacking is a form of ensemble learning [10] in which a learning algorithm is trained 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 [9] is represented as

$$\tilde{y}_{i+1} = \phi(\tilde{y}_{i+1,1}, \tilde{y}_{i+1,2}, \ldots, \tilde{y}_{i+1,m}, \sum_{j=1}^{m} \epsilon_{i+1,j}) \tag{7}$$

For a simple ϕ such as the ensemble average then equation 7 becomes

$$\tilde{y}_{i+1} = \frac{\tilde{y}_{i+1,1}, \tilde{y}_{i+1,2}, \ldots, \tilde{y}_{i+1,m}}{m} \tag{8}$$

In section 9.1 the simple model average is used to show how stacked forecast models are built in **mlr**. Section 9.2 does a more advanced method of ensemble averaging involving the forecast of endogeneous variables.

9.1 Stacking Univariate Learners

For this example the models TBATS, GARCH, and ARFIMA [21] are stacked together and averaged on the climate task data. A resample description is made, and the function `makeLearners()` is used to start multiple 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 **??**, however a super learner [43] 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. This leads to longer code, but tuning over the cross product of the models allows for a more honest perspective of how each model interacts in the stack.

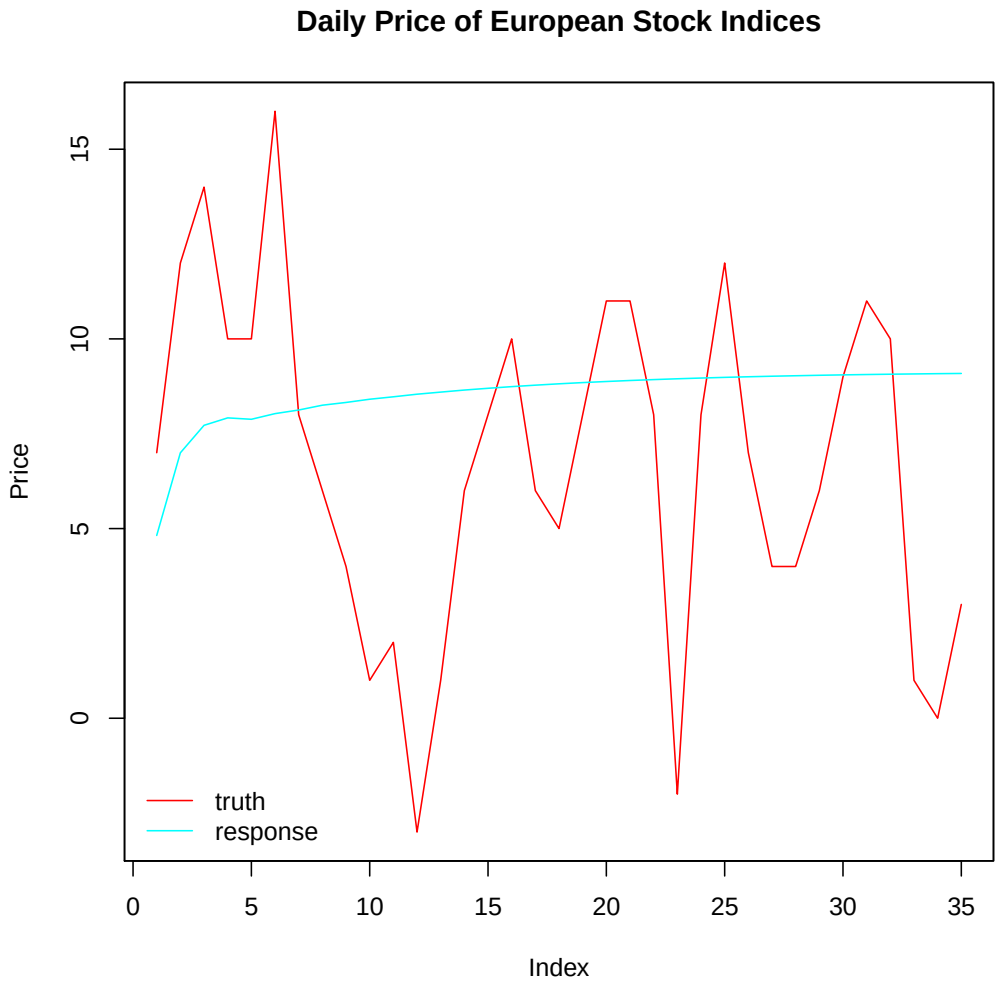
```
# Simple param set for tuning sub learners
ps = makeParamSet(
  makeDiscreteParam("fcregr.tbats.h", values = 35),
  makeDiscreteParam("fcregr.garch.n.ahead", values = 35),
  makeDiscreteParam("fcregr.arfima.h", values = 35),
  makeDiscreteParam("fcregr.arfima.estim", values = "ls"),
  makeDiscreteParam(id = "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 are handled 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
```



9.2 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 7 can be modified to include forecasts of other predictors x_i, k where k is the index for each predictor variable

$$\tilde{y}_{i+1} = \phi(\tilde{y}_{i+1,1}, \ldots, \tilde{y}_{i+1,m}, \tilde{x}_{i+1,1}, \ldots, \tilde{x}_{i+1,k}, \sum_{j=1}^{m} \epsilon_{i+1,j}, \sum_{j=1}^{k} \epsilon_{i+1,j}) \tag{9}$$

In the example below, a boosted glm [6] 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()`.

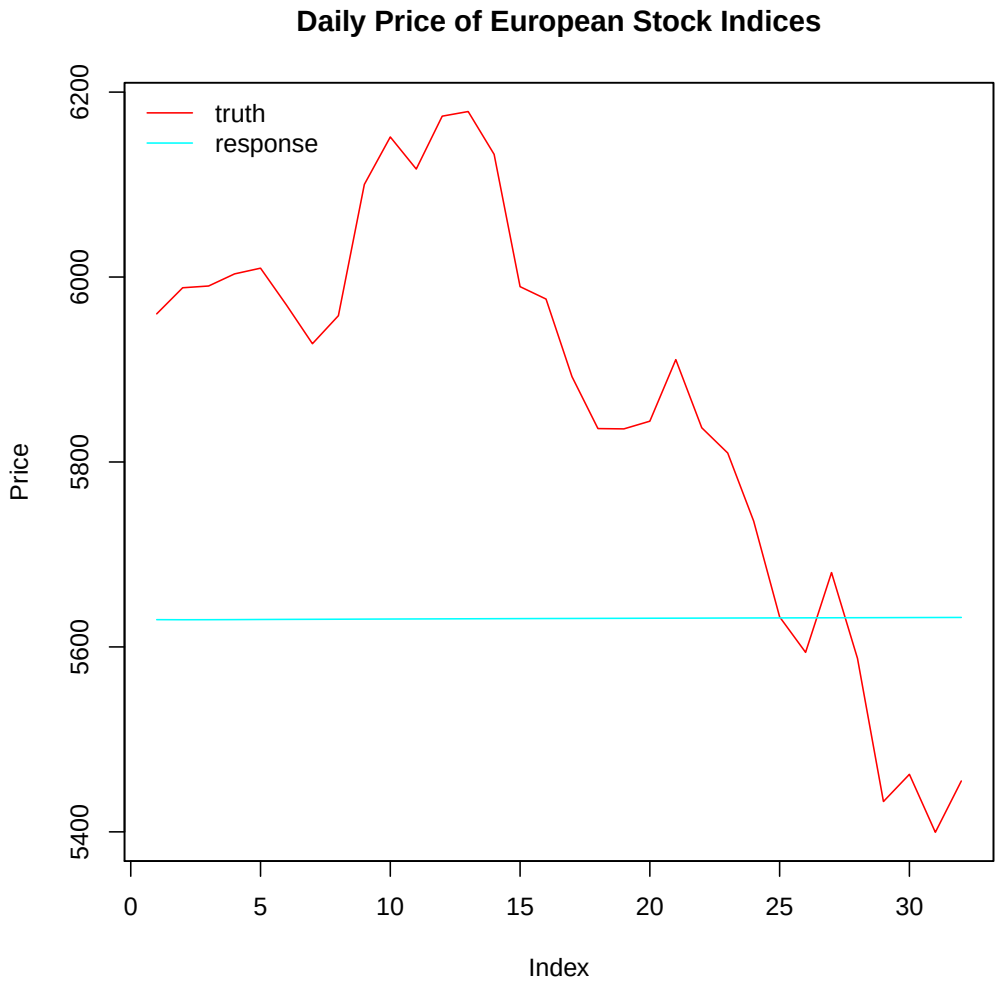
```
ps = makeParamSet(
  makeDiscreteParam("mfcregr.BigVAR.p", values = 9),
  makeDiscreteParam("mfcregr.BigVAR.struct", values = "SparseLag"),
  makeNumericVectorParam("mfcregr.BigVAR.gran", len = 2L, lower = 35,
                         upper = 50),
  makeDiscreteParam("mfcregr.BigVAR.h", values =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

```
## Tune result:
## Op. pars: mfcregr.BigVAR.p=5; mfcregr.BigVAR.struct=SparseLag; mfcregr.BigVAR.g
## multivar.mase.test.mean=0.453
```

Once the tuning is complete, the best model can be extracted with `setHyperPar2()` and the final model will be trained on all of the data. Since we are using the multivariate model to produce forecasts for an singel variable, univariate MASE is use 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
```



10 Conclusion

The results of this paper show that creating a unified interface for forecasting models in R allows for better models through an automated methedology of resampling, preprocessing, model selection, stacking tuning, and training. Building on the wide 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 beauracracy of modeling and more time testing new models. New methods such as multivariate stacked learners, Lambert W transforms, and the ability to create arbitary AR(p, d) machine learning models allows researchers to easily experiment with new ideas. 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.

Multivariate stacked learners are quite new, and to this researchers knowledge have not been used in this context. Future research based on this package would involve tuning these models to see how useful they are in the real world. Classification forecast made available in this extension is a very new field, and with the ease of making these models new research in this area can progress much quicker. Updates to this package will include more multivariate and univariate forecast learners as well as new methods to stack models such as Bayesian averaging [38].

References

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

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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.

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

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

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

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

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

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

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

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

[37] 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.

[38] 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.

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

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

[41] 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.

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

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

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