`modeltime` is a new package designed for rapidly developing and testing time series models using machine learning models, classical models, and automated models. There are three key benefits:

1. **Systematic Workflow for Forecasting.** Learn a few key functions like `modeltime_table()`, `modeltime_calibrate()`, and `modeltime_refit()` to develop and train time series models.
2. **Unlocks Tidymodels for Forecasting.** Gain the benefit of all or the `parsnip` models including `boost_tree()` (XGBoost, C5.0), `linear_reg()` (GLMnet, Stan, Linear Regression), `rand_forest()` (Random Forest), and more
3. **New Time Series Boosted Models** including Boosted ARIMA (`arima_boost()`) and Boosted Prophet (`prophet_boost()`) that can improve accuracy by applying XGBoost model to the errors

Getting Started
Let's kick the tires on modeltime

Install `modeltime`.

```
install.packages("modeltime")
```

Load the following libraries.

```
library(tidyverse)
library(tidymodels)
library(modeltime)
library(timetk)
library(lubridate)
```

Get Your Data
Forecasting daily bike transactions

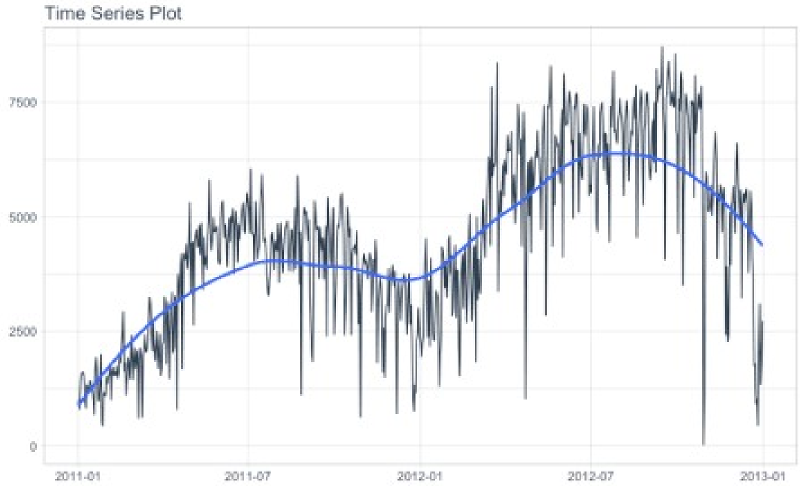
We'll start with a `bike_sharing_daily` time series data set that includes bike transactions. We'll simplify the data set to a univariate time series with columns, "date" and "value".

```
bike_transactions_tbl <- bike_sharing_daily %>%
  select(dteday, cnt) %>%
  set_names(c("date", "value"))

bike_transactions_tbl
```

```
## # A tibble: 731 x 2
##    date       value
##
##  1 2011-01-01   985
##  2 2011-01-02   801
##  3 2011-01-03  1349
##  4 2011-01-04  1562
##  5 2011-01-05  1600
##  6 2011-01-06  1606
##  7 2011-01-07  1510
##  8 2011-01-08   959
##  9 2011-01-09   822
## 10 2011-01-10  1321
## # … with 721 more rows
```

Next, visualize the dataset with the `plot_time_series()` function. Toggle `.interactive = TRUE` to get a plotly interactive plot. `FALSE` returns a ggplot2 static plot.

```
bike_transactions_tbl %>%
  plot_time_series(date, value, .interactive = FALSE)
```



Train / Test
Split your time series into training and testing sets

Next, use `time_series_split()` to make a train/test set.

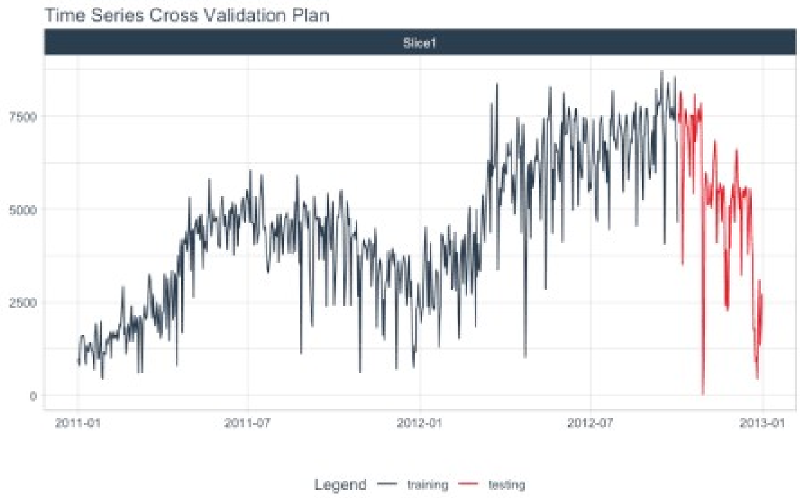
- Setting `assess = "3 months"` tells the function to use the last 3-months of data as the testing set.
- Setting `cumulative = TRUE` tells the sampling to use all of the prior data as the training set.

```
splits <- bike_transactions_tbl %>%
  time_series_split(assess = "3 months", cumulative = TRUE)
```

Next, visualize the train/test split.

- `tk_time_series_cv_plan()`: Converts the splits object to a data frame
- `plot_time_series_cv_plan()`: Plots the time series sampling data using the "date" and "value" columns.

```
splits %>%
  tk_time_series_cv_plan() %>%
  plot_time_series_cv_plan(date, value, .interactive = FALSE)
```



Modeling
This is **exciting.**

Now for the fun part! Let's make some models using functions from `modeltime` and `parsnip`.

Automatic Models

Automatic models are generally modeling approaches that have been automated. This includes "Auto ARIMA" and "Auto ETS" functions from `forecast` and the "Prophet" algorithm from `prophet`. These algorithms have been integrated into `modeltime`. The process is simple to set up:

- **Model Spec:** Use a specification function (e.g. `arima_reg()`, `prophet_reg()`) to initialize the algorithm and key parameters
- **Engine:** Set an engine using one of the engines available for the Model Spec.
- **Fit Model**: Fit the model to the training data

Let's make several models to see this process in action.

Auto ARIMA

Here's the basic Auto Arima Model fitting process.

- **Model Spec: `arima_reg()`** <– This sets up your general model algorithm and key parameters
- **Set Engine: `set_engine("auto_arima")`** <– This selects the specific package-function to use and you can add any function-level arguments here.
- **Fit Model: `fit(value ~ date, training(splits))`** <– All modeltime models require a date column to be a regressor.

```
model_fit_arima <- arima_reg() %>%
  set_engine("auto_arima") %>%
  fit(value ~ date, training(splits))
```

```
## frequency = 7 observations per 1 week
```

```
model_fit_arima
```

```
## parsnip model object
##
## Fit time:  326ms
## Series: outcome
## ARIMA(0,1,3) with drift
##
## Coefficients:
##           ma1      ma2      ma3   drift
##       -0.6106  -0.1868  -0.0673  9.3169
## s.e.   0.0396   0.0466   0.0398  4.6225
##
## sigma^2 estimated as 730568:  log likelihood=-5227.22
## AIC=10464.44   AICc=10464.53   BIC=10486.74
```

Prophet

Prophet is specified just like Auto ARIMA. Note that I've changed to `prophet_reg()`, and I'm passing an engine-specific parameter `yearly.seasonality = TRUE` using `set_engine()`.

```
model_fit_prophet <- prophet_reg() %>%
  set_engine("prophet", yearly.seasonality = TRUE) %>%
  fit(value ~ date, training(splits))
```

```
model_fit_prophet
```

```
## parsnip model object
##
```

```
## Fit time:  146ms
## PROPHET Model
## - growth: 'linear'
## - n.changepoints: 25
## - seasonality.mode: 'additive'
## - extra_regressors: 0
```

Machine Learning Models

Machine learning models are more complex than the automated models. This complexity typically requires a ***workflow*** (sometimes called a *pipeline* in other languages). The general process goes like this:

- **Create Preprocessing Recipe**
- **Create Model Specifications**
- **Use Workflow to combine Model Spec and Preprocessing, and Fit Model**

Preprocessing Recipe

First, I'll create a preprocessing recipe using `recipe()` and adding time series steps. The process uses the "date" column to create 45 new features that I'd like to model. These include time-series signature features and fourier series.

```
recipe_spec <- recipe(value ~ date, training(splits)) %>%
  step_timeseries_signature(date) %>%
  step_rm(contains("am.pm"), contains("hour"), contains("minute"),
          contains("second"), contains("xts")) %>%
  step_fourier(date, period = 365, K = 5) %>%
  step_dummy(all_nominal())

recipe_spec %>% prep() %>% juice()
```

```
## # A tibble: 641 x 47
##    date       value date_index.num date_year date_year.iso
date_half
##
##  1 2011-01-01   985     1293840000      2011          2010
1
##  2 2011-01-02   801     1293926400      2011          2010
1
##  3 2011-01-03  1349     1294012800      2011          2011
1
##  4 2011-01-04  1562     1294099200      2011          2011
1
##  5 2011-01-05  1600     1294185600      2011          2011
1
##  6 2011-01-06  1606     1294272000      2011          2011
1
##  7 2011-01-07  1510     1294358400      2011          2011
1
##  8 2011-01-08   959     1294444800      2011          2011
1
##  9 2011-01-09   822     1294531200      2011          2011
1
## 10 2011-01-10  1321     1294617600      2011          2011
1
## # … with 631 more rows, and 41 more variables: date_quarter ,
## #   date_month , date_day , date_wday , date_mday ,
## #   date_qday , date_yday , date_mweek , date_week ,
## #   date_week.iso , date_week2 , date_week3 , date_week4 ,
```

```
## #   date_mday7 , date_sin365_K1 , date_cos365_K1 ,
## #   date_sin365_K2 , date_cos365_K2 , date_sin365_K3 ,
## #   date_cos365_K3 , date_sin365_K4 , date_cos365_K4 ,
## #   date_sin365_K5 , date_cos365_K5 , date_month.lbl_01 ,
## #   date_month.lbl_02 , date_month.lbl_03 , date_month.lbl_04 ,
## #   date_month.lbl_05 , date_month.lbl_06 , date_month.lbl_07 ,
## #   date_month.lbl_08 , date_month.lbl_09 , date_month.lbl_10 ,
## #   date_month.lbl_11 , date_wday.lbl_1 , date_wday.lbl_2 ,
## #   date_wday.lbl_3 , date_wday.lbl_4 , date_wday.lbl_5 ,
## #   date_wday.lbl_6
```

With a recipe in-hand, we can set up our machine learning pipelines.

Elastic Net

Making an Elastic NET model is easy to do. Just set up your model spec using `linear_reg()` and `set_engine("glmnet")`. Note that we have not fitted the model yet (as we did in previous steps).

```
model_spec_glmnet <- linear_reg(penalty = 0.01, mixture = 0.5) %>%
  set_engine("glmnet")
```

Next, make a fitted workflow:

- **Start** with a `workflow()`
- **Add a Model Spec:** `add_model(model_spec_glmnet)`
- **Add Preprocessing:** `add_recipe(recipe_spec %>% step_rm(date))` <– Note that I'm removing the "date" column since Machine Learning algorithms don't typically know how to deal with date or date-time features
- **Fit the Workflow**: `fit(training(splits))`

```
workflow_fit_glmnet <- workflow() %>%
  add_model(model_spec_glmnet) %>%
  add_recipe(recipe_spec %>% step_rm(date)) %>%
  fit(training(splits))
```

Random Forest

We can fit a Random Forest using a similar process as the Elastic Net.

```
model_spec_rf <- rand_forest(trees = 500, min_n = 50) %>%
  set_engine("randomForest")

workflow_fit_rf <- workflow() %>%
  add_model(model_spec_rf) %>%
  add_recipe(recipe_spec %>% step_rm(date)) %>%
  fit(training(splits))
```

New Hybrid Models

I've included several hybrid models (e.g. `arima_boost()` and `prophet_boost()`) that combine both automated algorithms with machine learning. I'll showcase `prophet_boost()` next!

Prophet Boost

The ***Prophet Boost algorithm*** combines Prophet with XGBoost to get the best of both worlds (i.e. Prophet Automation + Machine Learning). The algorithm works by:

1. First modeling the univariate series using Prophet
2. Using regressors supplied via the preprocessing recipe (remember our recipe generated 45 new features), and regressing the Prophet Residuals with the XGBoost model

We can set the model up using a workflow just like with the machine learning algorithms.

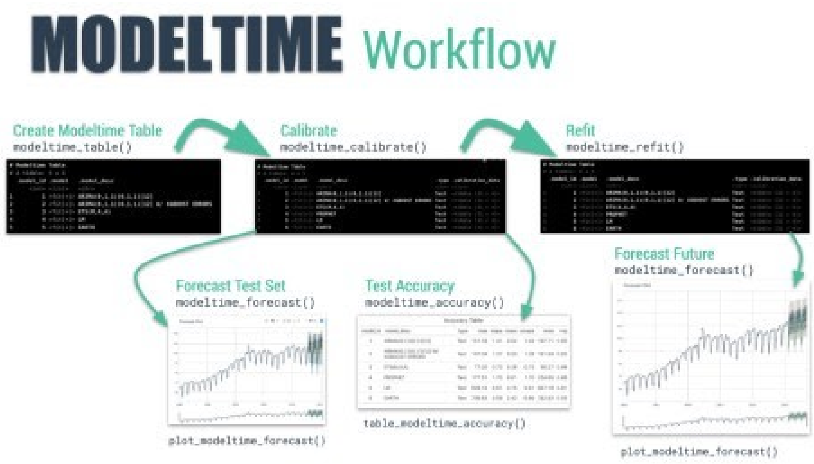
```
model_spec_prophet_boost <- prophet_boost() %>%
  set_engine("prophet_xgboost", yearly.seasonality = TRUE)

workflow_fit_prophet_boost <- workflow() %>%
  add_model(model_spec_prophet_boost) %>%
  add_recipe(recipe_spec) %>%
  fit(training(splits))
```

```
## [07:25:50] WARNING: amalgamation/../src/learner.cc:480:
## Parameters: { validation } might not be used.
##
##   This may not be accurate due to some parameters are only used in
language bindings but
##   passed down to XGBoost core.  Or some parameters are not used but
slip through this
##   verification. Please open an issue if you find above cases.
```

```
workflow_fit_prophet_boost
```

```
## ══ Workflow [trained] ══════════════════════════════
════════════════════════════════════════════════════════════
════════════════
## Preprocessor: Recipe
## Model: prophet_boost()
##
## ── Preprocessor ──────────────────────────────
────────────────────────────────────────────────────────────
──────────────────────
## 4 Recipe Steps
##
## ● step_timeseries_signature()
## ● step_rm()
## ● step_fourier()
## ● step_dummy()
##
## ── Model ──────────────────────────────
────────────────────────────────────────────────────────────
─────────────────────────────
## PROPHET w/ XGBoost Errors
## ---
## Model 1: PROPHET
## - growth: 'linear'
## - n.changepoints: 25
## - seasonality.mode: 'additive'
##
## ---
## Model 2: XGBoost Errors
##
## xgboost::xgb.train(params = list(eta = 0.3, max_depth = 6, gamma =
0,
##     colsample_bytree = 1, min_child_weight = 1, subsample = 1),
##     data = x, nrounds = 15, verbose = 0, early_stopping_rounds =
NULL,
##     objective = "reg:squarederror", validation = 0, nthread = 1)
```

The Modeltime Workflow
Speed up model evaluation and selection with modeltime



The `modeltime` workflow is designed to speed up model evaluation and selection. Now that we have several time series models, let's analyze them and forecast the future with the `modeltime` workflow.

Modeltime Table

The Modeltime Table organizes the models with IDs and creates generic descriptions to help us keep track of our models. Let's add the models to a `modeltime_table()`.

```
model_table <- modeltime_table(
  model_fit_arima,
  model_fit_prophet,
  workflow_fit_glmnet,
  workflow_fit_rf,
  workflow_fit_prophet_boost
)

model_table

## # Modeltime Table
## # A tibble: 5 x 3
##   .model_id .model     .model_desc
##
## 1         1   ARIMA(0,1,3) WITH DRIFT
## 2         2   PROPHET
## 3         3 GLMNET
## 4         4 RANDOMFOREST
## 5         5 PROPHET W/ XGBOOST ERRORS
```

Calibration

Model Calibration is used to quantify error and estimate confidence intervals. We'll perform model calibration on the out-of-sample data (aka. the Testing Set) with the `modeltime_calibrate()` function. Two new columns are generated (".type" and ".calibration_data"), the most important of which is the ".calibration_data". This includes the actual values, fitted values, and residuals for the testing set.

```
calibration_table <- model_table %>%
  modeltime_calibrate(testing(splits))

calibration_table

## # Modeltime Table
## # A tibble: 5 x 5
##   .model_id .model     .model_desc                  .type
.calibration_data
##
```

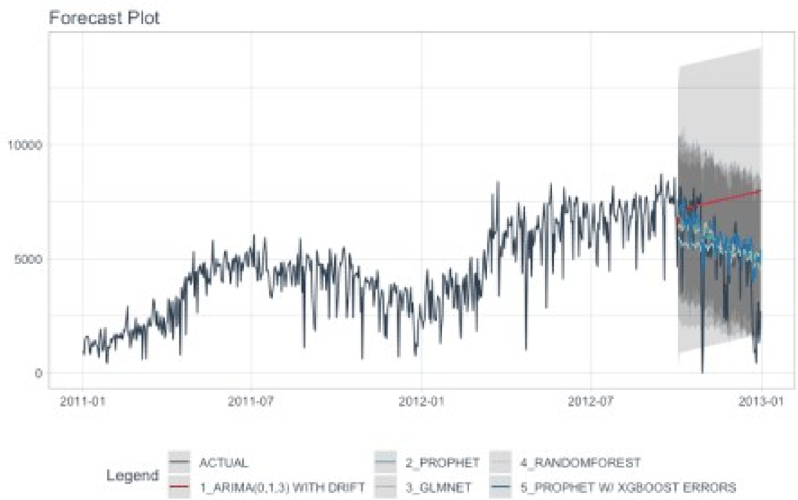
```
## 1          1    ARIMA(0,1,3) WITH DRIFT   Test
## 2          2    PROPHET                   Test
## 3          3  GLMNET                    Test
## 4          4  RANDOMFOREST              Test
## 5          5  PROPHET W/ XGBOOST ERRORS Test
```

Forecast (Testing Set)

With calibrated data, we can visualize the testing predictions (forecast).

- Use `modeltime_forecast()` to generate the forecast data for the testing set as a tibble.
- Use `plot_modeltime_forecast()` to visualize the results in interactive and static plot formats.

```
calibration_table %>%
  modeltime_forecast(actual_data = bike_transactions_tbl) %>%
  plot_modeltime_forecast(.interactive = FALSE)
```



Accuracy (Testing Set)

Next, calculate the testing accuracy to compare the models.

- Use `modeltime_accuracy()` to generate the out-of-sample accuracy metrics as a tibble.
- Use `table_modeltime_accuracy()` to generate interactive and static

```
calibration_table %>%
  modeltime_accuracy() %>%
  table_modeltime_accuracy(.interactive = FALSE)
```

Accuracy Table

.model_id	.model_desc	.type	mae	mape	mase	smape	rmse	rsq
1	ARIMA(0,1,3) WITH DRIFT	Test	2540.11	474.89	2.74	46.00	3188.09	0.39
2	PROPHET	Test	1221.18	365.13	1.32	28.68	1764.93	0.44
3	GLMNET	Test	1197.06	340.57	1.29	28.44	1650.87	0.49
4	RANDOMFOREST	Test	1338.15	335.52	1.45	30.63	1855.21	0.46
5	PROPHET W/ XGBOOST ERRORS	Test	1189.28	332.44	1.28	28.48	1644.25	0.55

Analyze Results

From the accuracy measures and forecast results, we see that:

- Auto ARIMA model is not a good fit for this data.
- The best model is Prophet + XGBoost

Let's exclude the Auto ARIMA from our final model, then make future forecasts with the remaining models.

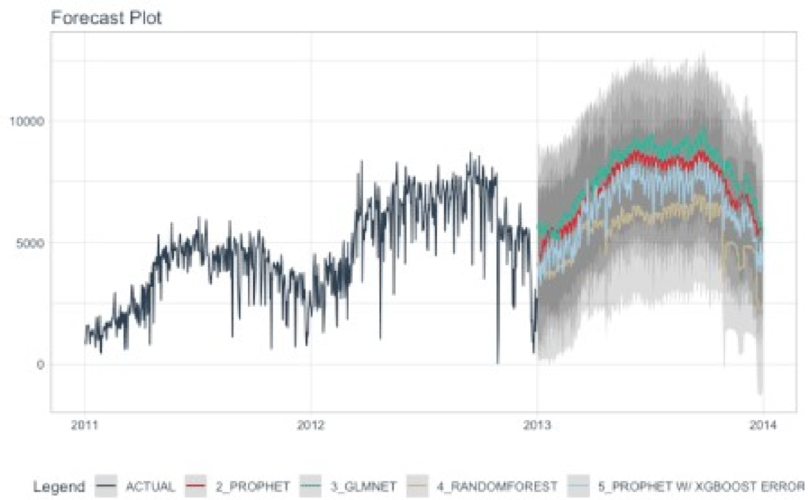
Refit and Forecast Forward

Refitting is a best-practice before forecasting the future.

- `modeltime_refit()`: We re-train on full data (`bike_transactions_tbl`)
- `modeltime_forecast()`: For models that only depend on the "date" feature, we can use `h` (horizon) to forecast forward. Setting `h = "12 months"` forecasts then next 12-months of data.

```
calibration_table %>%
  # Remove ARIMA model with low accuracy
  filter(.model_id != 1) %>%

  # Refit and Forecast Forward
  modeltime_refit(bike_transactions_tbl) %>%
  modeltime_forecast(h = "12 months", actual_data =
bike_transactions_tbl) %>%
  plot_modeltime_forecast(.interactive = FALSE)
```

```
## [07:25:57] WARNING: amalgamation/../src/learner.cc:480:
## Parameters: { validation } might not be used.
##
##   This may not be accurate due to some parameters are only used in
language bindings but
##   passed down to XGBoost core.  Or some parameters are not used but
slip through this
##   verification. Please open an issue if you find above cases.
```



It gets better