Forecasting Daily Air Quality Index of Delhi

Ankita Khandelwal, Pranav Jain, Siddharth Dixit

Introduction

As the impact of air pollutants on human health through ambient air has addressed much attention in recent years, the air quality forecasting in terms of air pollution parameters becomes an important topic in environmental science.
Monitoring air and understanding its quality is of immense importance to our well-being. The standard measure for the quality of air is the Air Quality Index (AQI), which can be estimated through a formula, based on comprehensive assessment of concentration of air pollutants. This is used by government agencies to characterize the status of air quality at a given location. The present study aims at developing forecasting models for predicting daily AQI measures, which can be used as a basis for the decision making processes.

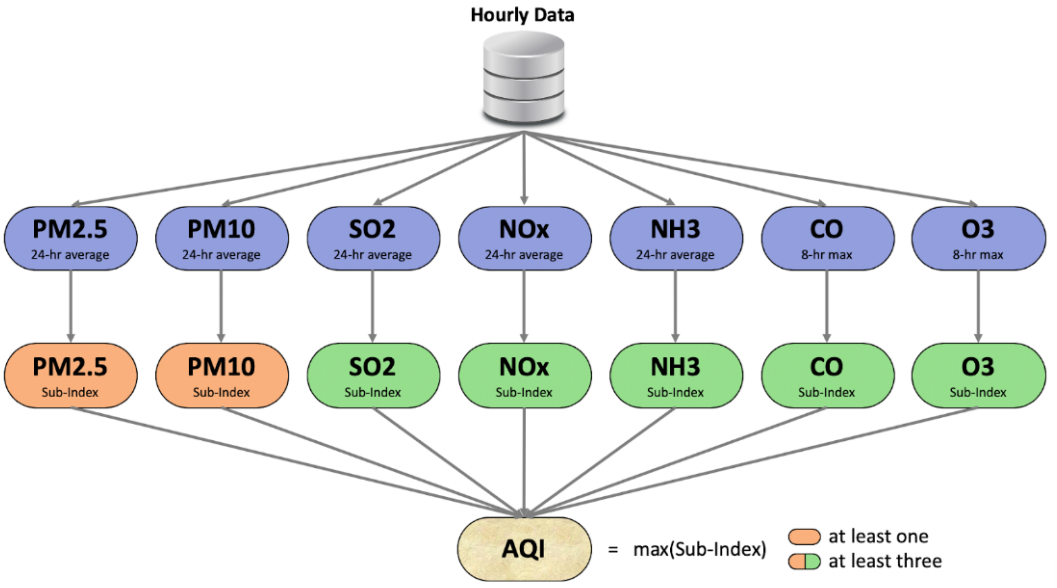


Figure 1: Formula for AQI Calculation

- The AQI calculation uses 7 measures: PM2.5, PM10, SO2, NOx, NH3, CO and O3.
- For PM2.5, PM10, SO2, NOx and NH3, the average value in the last 24-hrs is used with the condition of having at least 16 values.
- For CO and O3, the maximum value in the last 8-hrs is used.
- Each measure is converted into a Sub-Index based on pre-defined groups.
- Sometimes measures are not available due to lack of measuring or lack of required data points.
- Final AQI is the maximum Sub-Index with the condition that at least one of PM2.5 and PM10 should be available and at least three out of the seven should be available.

The pre-defined buckets of AQI are as follows

Good (0–50)	Minimal Impact	Poor (201–300)	Breathing discomfort to people on prolonged exposure
Satisfactory (51–100)	Minor breathing discomfort to sensitive people	Very Poor (301–400)	Respiratory illness to the people on prolonged exposure
Moderate (101–200)	Breathing discomfort to the people with lung, heart disease, children and older adults	Severe (>401)	Respiratory effects even on healthy people

Figure 2: AQI-Buckets

Relevance of AQI Forecasting

We know that when air quality is poor- it is unhealthy, especially for people who are sensitive to it such as children, older adults, or people with heart disease, asthma, and other respiratory ailments. Because different areas have different levels of air quality at different times, it is important for us to monitor what is happening. Through this,we can identify trouble spots and ensure that we are taking the right steps so that we all enjoy the cleanest air possible. Such forecasts can also help the government bodies in taking measures to keep its citizens informed and well prepared.

In this project, we are focusing on forecasting the Delhi AQI.

Data Availability

The data of AQI for each city has been made publicly available by the Central Pollution Control Board: https://cpcb.nic.in/ & https://www.kaggle.com/rohanrao/air-quality-data-in-india contains the cumulative India-wide data from Jan 2015- July 2020. This dataset contains air quality data and AQI at both hourly and daily levels. For the scope of this project, only the Delhi data is of concern to us.

Methodology

We describe our methodology in the following 3 parts:-

1) Exploratory Data Analysis

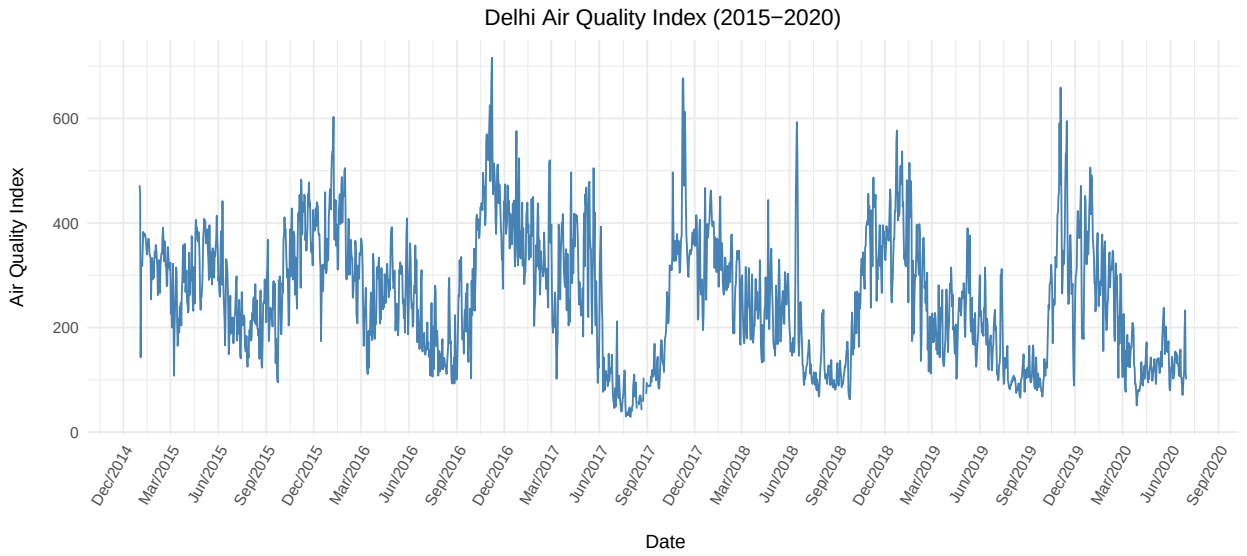
We start by loading some packages- *forecast, MLmetrics & ggplot2* which will be used as a part of our workflow. The dataset we will be working with, is stored in a excel file by the name of *'delhi_aqi_by_day'*.

```
library(forecast)
library(MLmetrics)
library(ggplot2)
library(readxl)
delhi_aqi_by_day <- read_excel("delhi_aqi_by_day.xlsx")
aqi_ts = delhi_aqi_by_day[c('Date','AQI')]
aqi_ts$Date = as.Date(aqi_ts$Date)
head(aqi_ts)# View the first 5 entries of the dataframe
```

```
## # A tibble: 6 x 2
##   Date         AQI
##   <date>     <dbl>
## 1 2015-01-01   472
## 2 2015-01-02   454
```

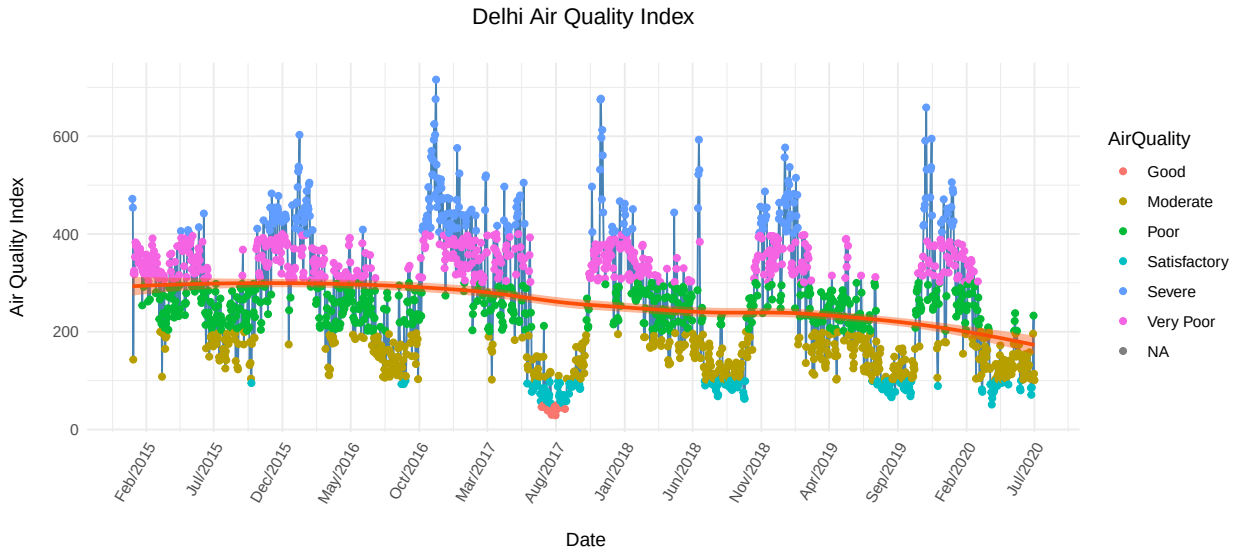
```
## 3 2015-01-03   143
## 4 2015-01-04   319
## 5 2015-01-05   325
## 6 2015-01-06   318
```

```
delhi_aqi_by_day$Date = as.Date(delhi_aqi_by_day$Date)
AirQuality = delhi_aqi_by_day$AQI_Bucket

# Normal Plot
p = ggplot(aqi_ts,aes(x = Date, y = AQI)) +
  geom_line(color="steelblue") +  # Blue lines
  #geom_point(aes(color=AirQuality))
  xlab("\n Date") +
  ylab("Air Quality Index\n")+
  theme_minimal()+
  theme(axis.text.x=element_text(angle=60, hjust=1))

p + scale_x_date(date_labels = "%b/%Y",date_breaks = '3 month')+
  ggtitle("Delhi Air Quality Index (2015-2020)")+
  theme(plot.title = element_text(hjust = 0.5))
```



In order to identify groups of similar AQI readings, we bucket the daily readings into 6 different categories.

```
## Bucketed Plot
p = ggplot(aqi_ts,aes(x = Date, y = AQI)) +
  geom_line(color="steelblue") +  # Blue lines
  geom_point(aes(color=AirQuality))+
  xlab("\n Date") +
  ylab("Air Quality Index\n")+
  theme_minimal()+
  theme(axis.text.x=element_text(angle=60, hjust=1))

p + scale_x_date(date_labels = "%b/%Y",date_breaks = '5 month')+ stat_smooth(
  color = "#FC4E07", fill = "#FC4E07",method = "loess")+
  ggtitle("Delhi Air Quality Index\n")+
  theme(plot.title = element_text(hjust = 0.5))
```

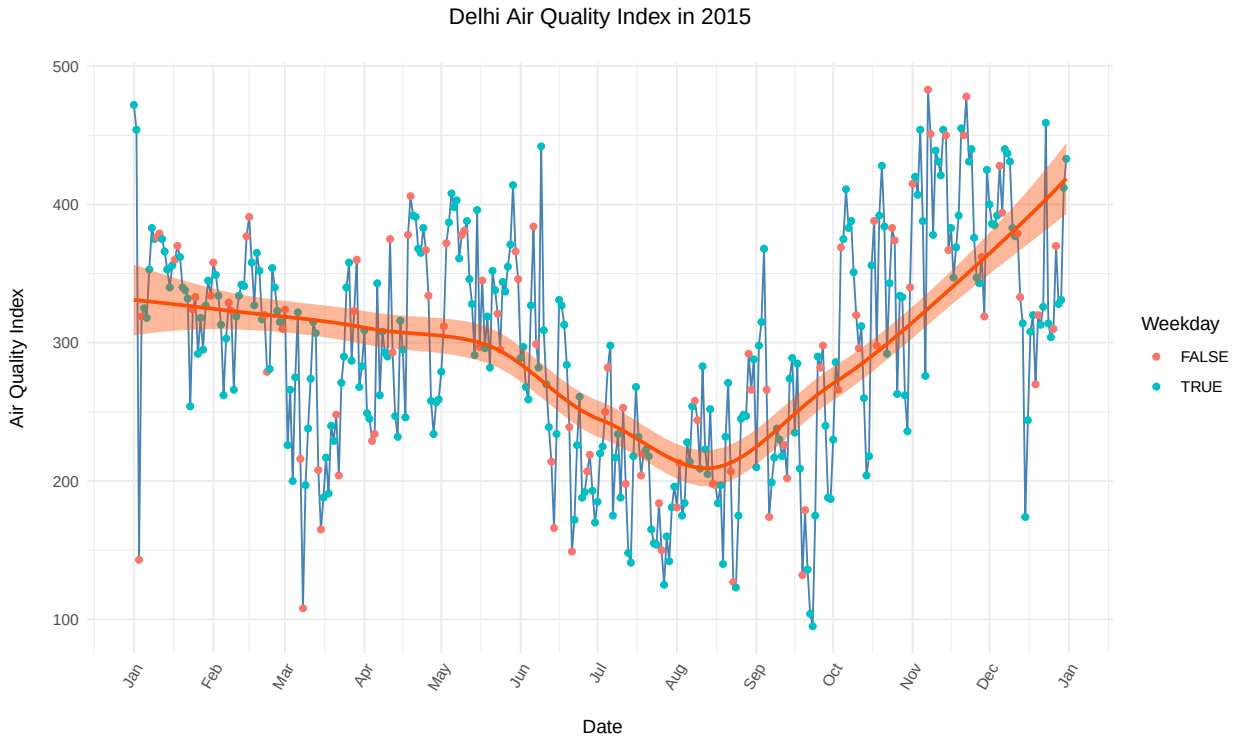


- The aqi is highest in the month of december and the lowest from June to August. Hence it can be inferred that the aqi drops drastically during the monsoon and rises suddenly during the harvest season.
- The orange line is the trend line for our data - we observe a decreasing trend following the 2017 monsoon

We now create an indicator variable to distinguish between weekdays and weekends. Furthermore, yearwise plots are plotted to see the how AQI has varied in the past five years on a monthly scale.

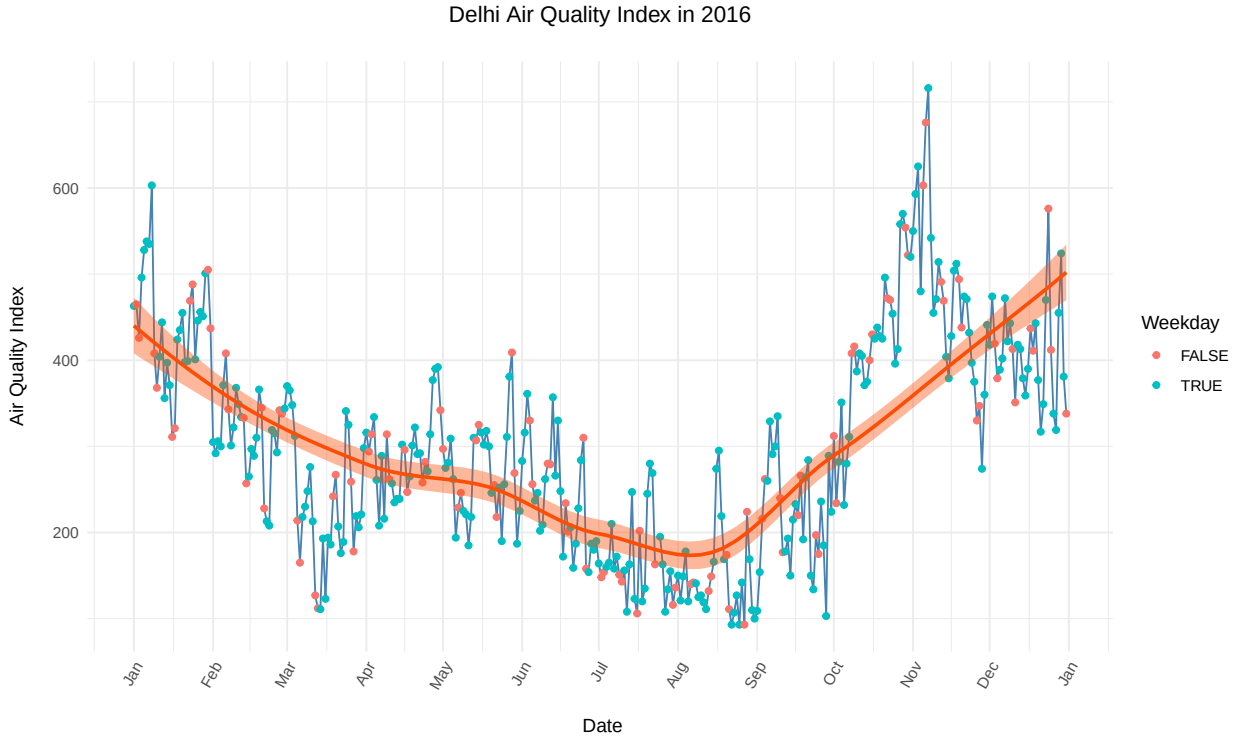
```
library(lubridate)
df_ts_plot = aqi_ts
df_ts_plot['Month'] = month(aqi_ts$Date, label = TRUE)
df_ts_plot['Year'] = year(aqi_ts$Date)
is_weekday = function(timestamp){
  lubridate::wday(timestamp, week_start = 1) < 6
}
weekday = is_weekday(df_ts_plot$Date)

df_ts_plot['Weekday'] = as.factor(weekday)
```

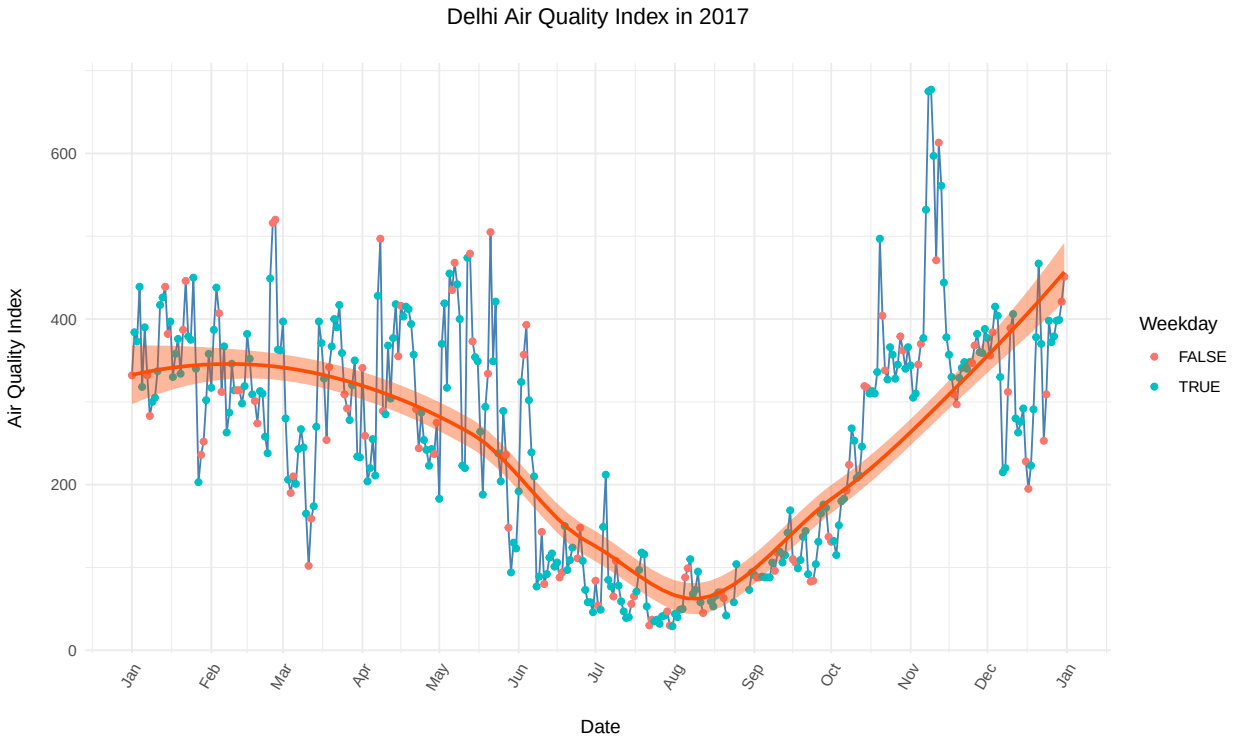
```
## For 2015
df_2015 = df_ts_plot[df_ts_plot$Year==2015,]

p = ggplot(df_2015,aes(x = Date, y = AQI)) +
  geom_line(color="steelblue") +  # Blue lines
  geom_point(aes(color=Weekday))+
  xlab("\n Date") +
  ylab("Air Quality Index\n")+
  theme_minimal()+
  theme(axis.text.x=element_text(angle=60, hjust=1))
p + scale_x_date(date_labels = "%b",date_breaks = '1 month')+ stat_smooth(
  color = "#FC4E07", fill = "#FC4E07",method = "loess")+
  ggtitle("Delhi Air Quality Index in 2015\n")+
  theme(plot.title = element_text(hjust = 0.5))
```

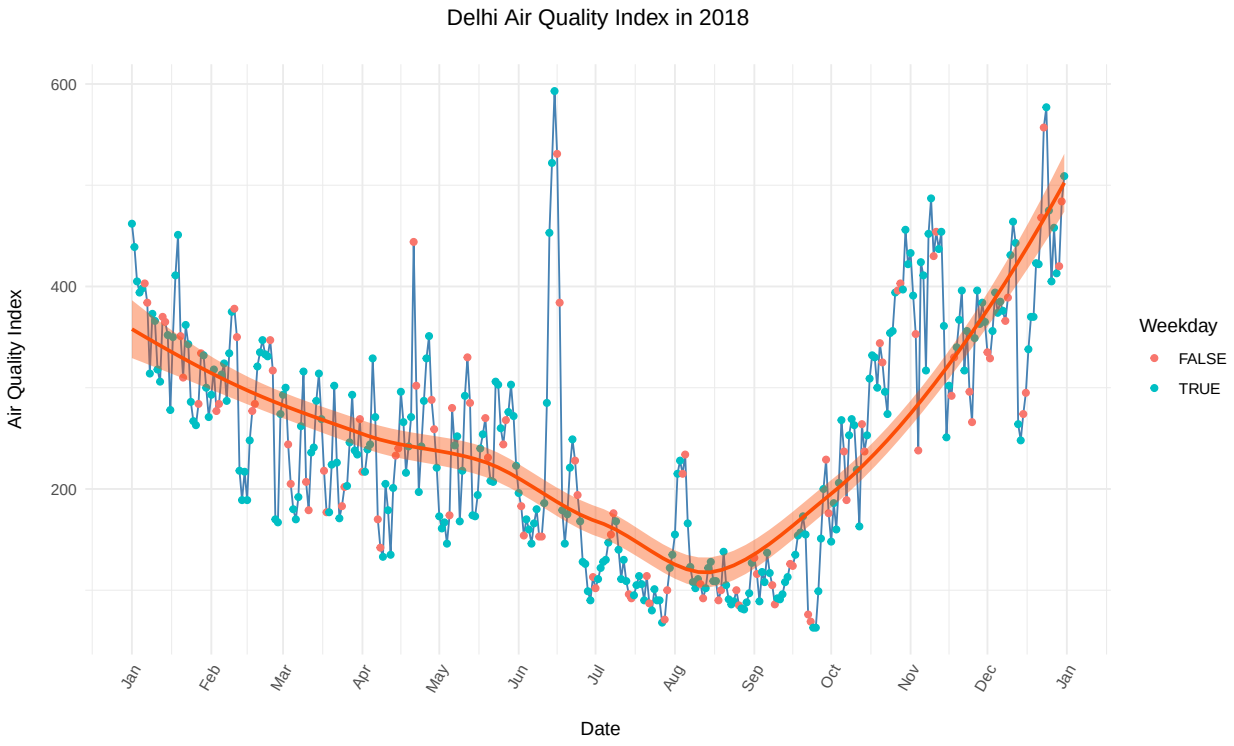


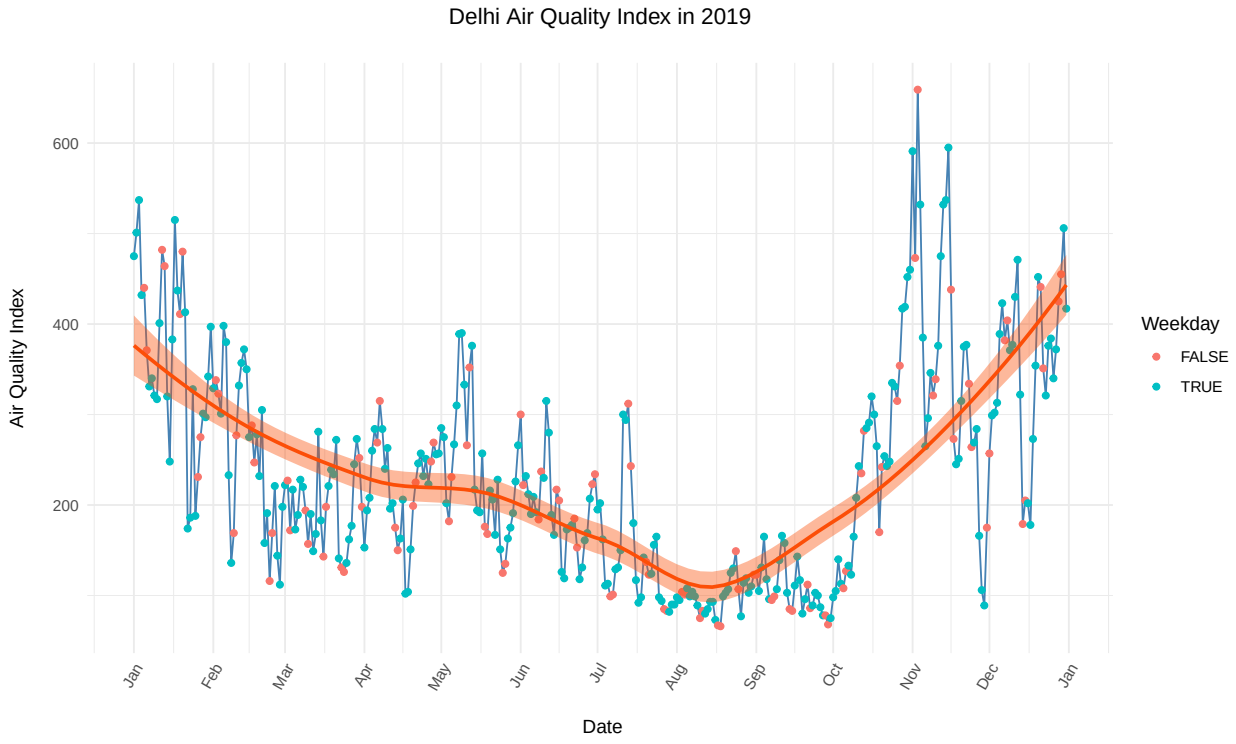
This is the timeseries plot for the year 2015. The decrease during the monsoons and the sudden rise during the harvest season is more apparent now (as we see in further graphs)

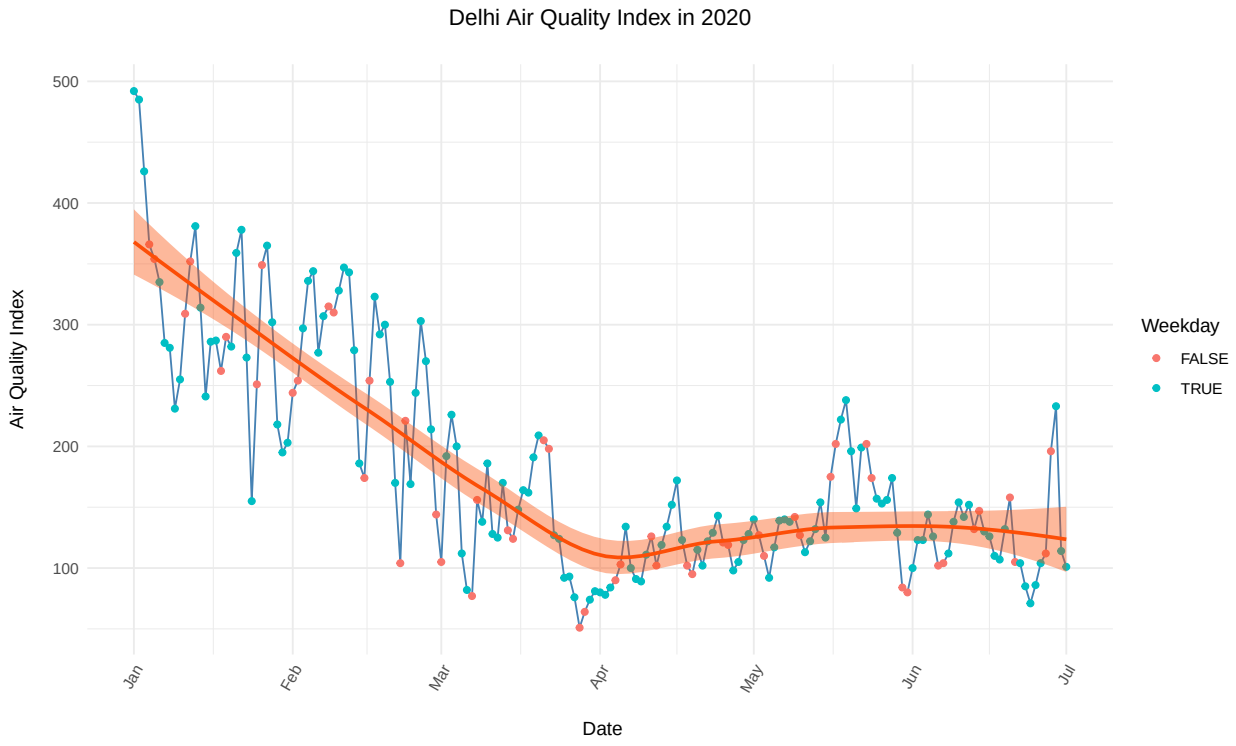
Making minor changes in the code used to generate the previous plot, we can produce similar plots for 2016-2020 as well.




Delhi Air Quality Index in 2017
Air Quality Index
600
400
200
0
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Date
Weekday
FALSE
TRUE


Delhi Air Quality Index in 2018
Air Quality Index
600
400
200
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Date
Weekday
FALSE
TRUE





The AQI dops under 100 towards the end of March and stays low into the summers. This is the same timeframe around which the lockdown for COVID19 was enforced. Hence it can be concluded that the reduction in activity due to the lockdown resulted in a reduction in the aqi.

Upon calculation, the AQI for thursday is greater than that on a Sunday by about 9.1 units on an average. Thus it is safe to say that the aqi levels are greater on weekdays compared to weekends.

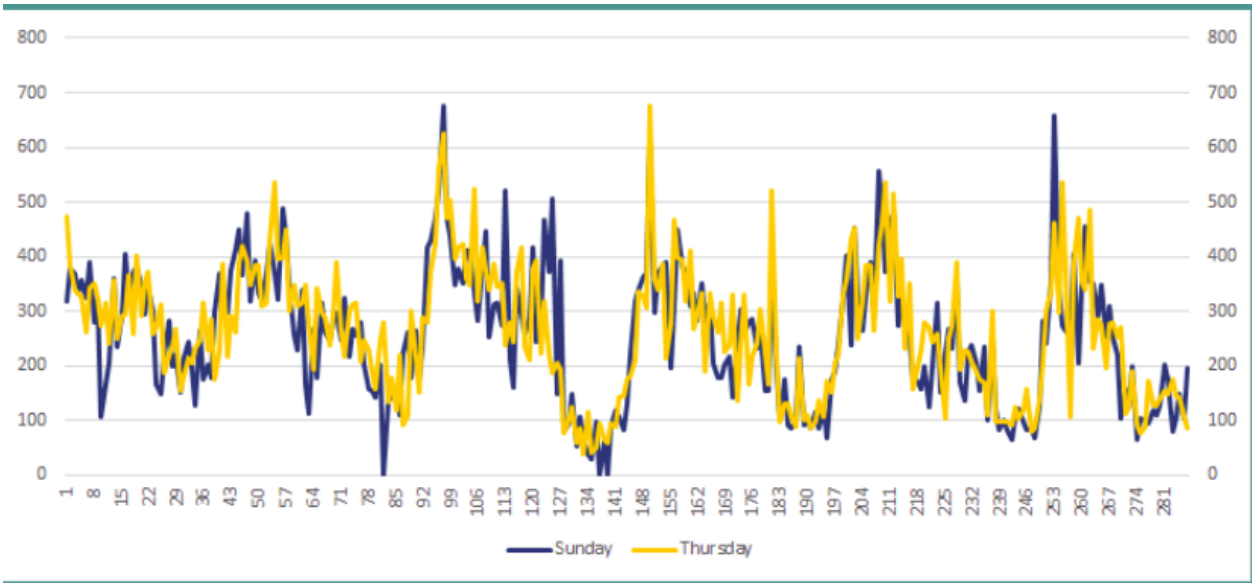


Figure 3: AQI on thursdays v/s sundays

2) Data Pre-processing

The daily air quality index dataset contained some missing values. So in order to impute them we followed the following strategy:-

- AQI value missing for a particular day was replaced by mean of previous day's and next day's AQI.
- If the previous/future day's AQI was also missing then they were replaced by the global mean.

The procedure has been implemented in the code below

```
df_ts <- ts(aqi_ts$AQI, start=c(2015,1), frequency=365)
#plot(df_ts, yax.flip = TRUE)

mean_aqi = mean(na.omit(aqi_ts$AQI))
c=1
for(val in df_ts){
  if(is.na(val)){
    if(!is.na(df_ts[c-1]) & !is.na(df_ts[c+1])){
      df_ts[c]= (df_ts[c-1]+df_ts[c+1])/2 # replace by mean of previous and next day's aqi
    }
    else{
      #replace by mean if previous or next day's aqi is missing
      df_ts[c]=mean_aqi
    }
  }
  c=c+1
}
# Double check if all NAs have been imputed correctly
sum(is.na(df_ts))
```

```
## [1] 0
```

We now store the cleaned version of the dataset in a different file for future use.

```
df_ts_new = aqi_ts
df_ts_new$AQI = df_ts
#write.csv(df_ts_new,"AQI_univar.csv")
```

Decomposition of Time Series

We now break down our daily AQI time series into a number of components.Time series decomposition involves thinking of a series as a combination of *level, trend, seasonality, and noise components*. These components are defined as follows:

- *Level*: The average value in the series.
- *Trend*: The increasing or decreasing value in the series.
- *Seasonality*: The repeating short-term cycle in the series.
- *Noise*: The random variation in the series.

Decomposition provides a useful abstract model for thinking about time series generally and for better understanding problems during time series analysis and forecasting.

It is helpful to think of these components as combining either **additively** or **multiplicatively**.

Additive Decomposition

An additive model suggests that the components are added together as follows:

$y(t) = Level + Trend + Seasonality + Noise$

An additive model is linear where changes over time are consistently made by the same amount. A linear trend is a straight line. A linear seasonality has the same frequency (width of cycles) and amplitude (height of cycles).

Multiplicative Decomposition

A multiplicative model suggests that the components are multiplied together as follows:

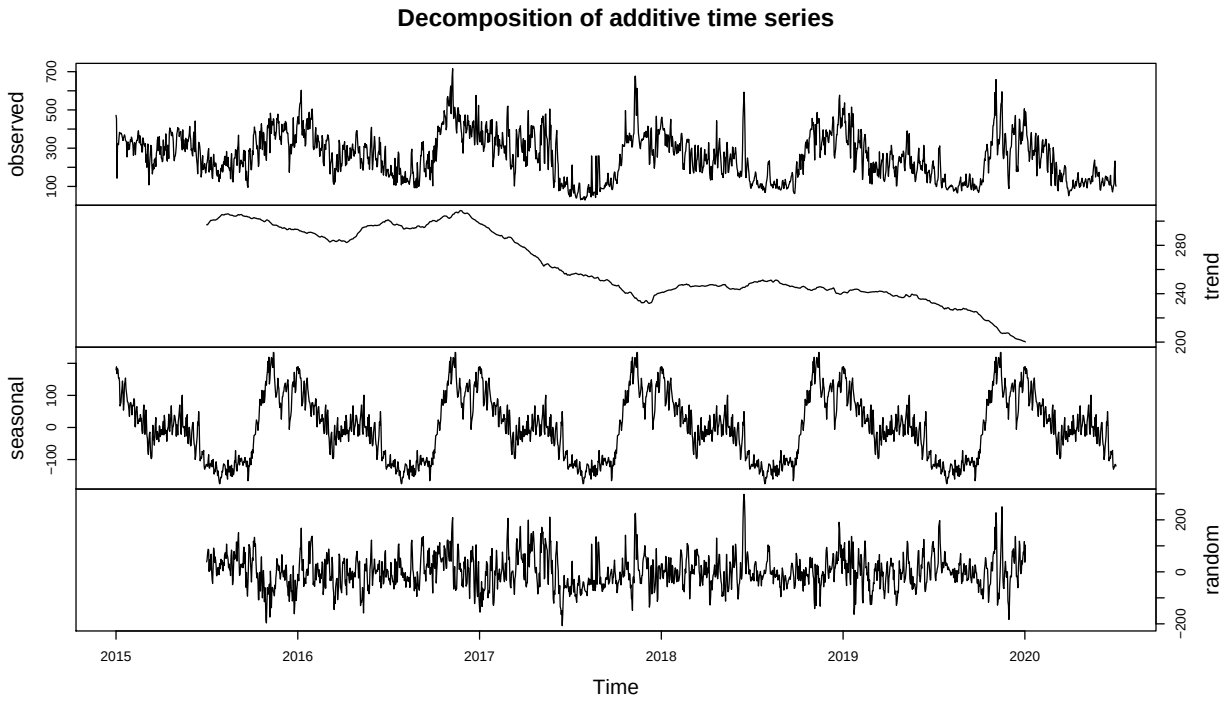
$y(t) = Level * Trend * Seasonality * Noise$

A multiplicative model is nonlinear, such as quadratic or exponential. Changes increase or decrease over time. A nonlinear trend is a curved line. A non-linear seasonality has an increasing or decreasing frequency and/or amplitude over time.

As a rule of thumb if seasonality of the time series tends to increase with time then the multiplicative model for decomposition is recommended otherwise if the seasonality remains constant then the additive model is used.

So as per the original time series plot, we can see that the seasonality variation seems to be more or less constant with time. Therefore an additive model will be more appropriate for the task.

```
# 1) Additive Decomposition
aqi.decomp <- decompose(df_ts,type = 'additive')
## plot the obs ts, trend & seasonal effect
plot(aqi.decomp, yax.flip = TRUE)
```




```
aqi_residuals = aqi.decomp$random
aqi_trend = aqi.decomp$trend
aqi_seasonality = aqi.decomp$seasonal
```

However, a problem while decomposing our AQI time series is that the decomposition procedure automatically introduces NA values at the start and end of the *trend & random* components.

```
head(aqi.decomp$trend,10)
```

```
## Time Series:
## Start = c(2015, 1)
## End = c(2015, 10)
## Frequency = 365
##  [1] NA NA NA NA NA NA NA NA NA NA
```

```
head(aqi.decomp$random,10)
```

```
## Time Series:
## Start = c(2015, 1)
## End = c(2015, 10)
## Frequency = 365
##  [1] NA NA NA NA NA NA NA NA NA NA
```

The reason behind this is evident from the documentation of the *decompose()* function itself, the trend component is estimated using a moving average with a symmetric window with equal weights. So the first and last few values used to calculate the moving average become NA by default.

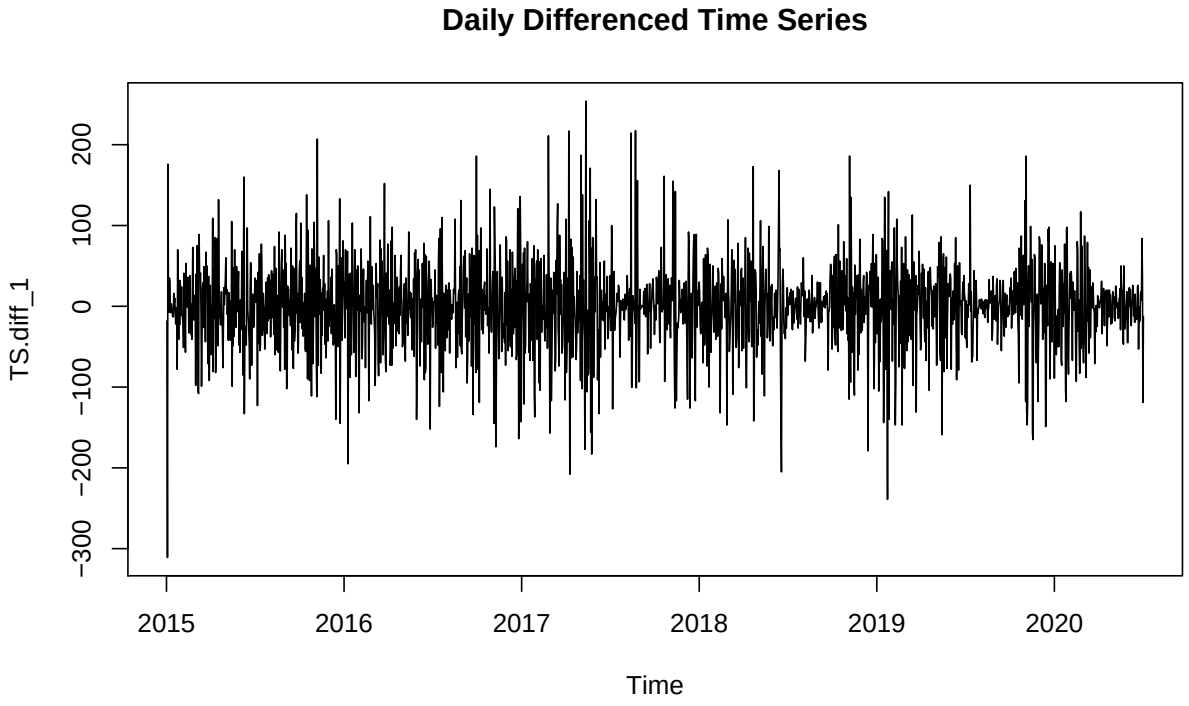
An alternative to decomposition for removing trends is differencing. The difference operator can be used to remove linear and nonlinear trends as well as various seasonal features that might be evident in the data. The difference operator is defined as:-

$\nabla x_t = x_t - x_{t-1}$ and more generally for order d as:-

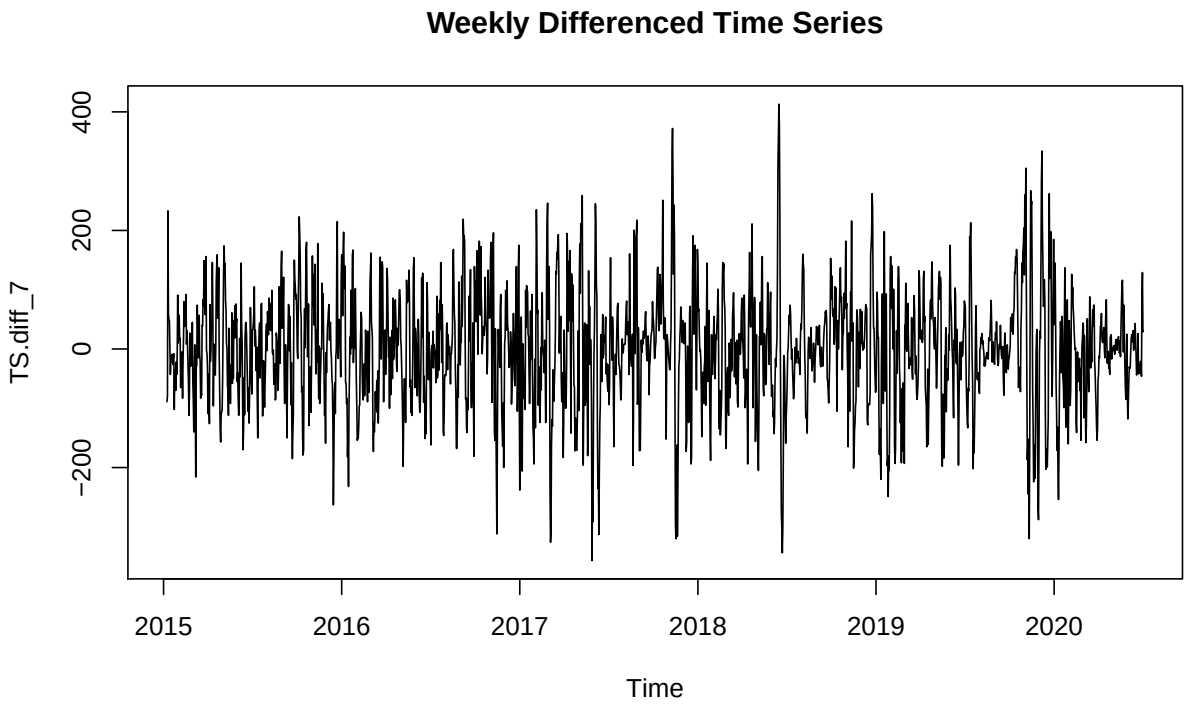
$\nabla^d x_t = (1-B)^d x_t$ where B is the backshift operator (i.e. $B^k x_t = x_{t-k}$ for $k \geq 1$)

In the code given below we perform two kinds of differencing viz. *daily & weekly lagged* and plot the differenced time series.

```
TS.diff_1 <- diff(df_ts,lag=1)
TS.diff_7 <- diff(df_ts,lag=7)
ts.plot(TS.diff_1, main="Daily Differenced Time Series")
```

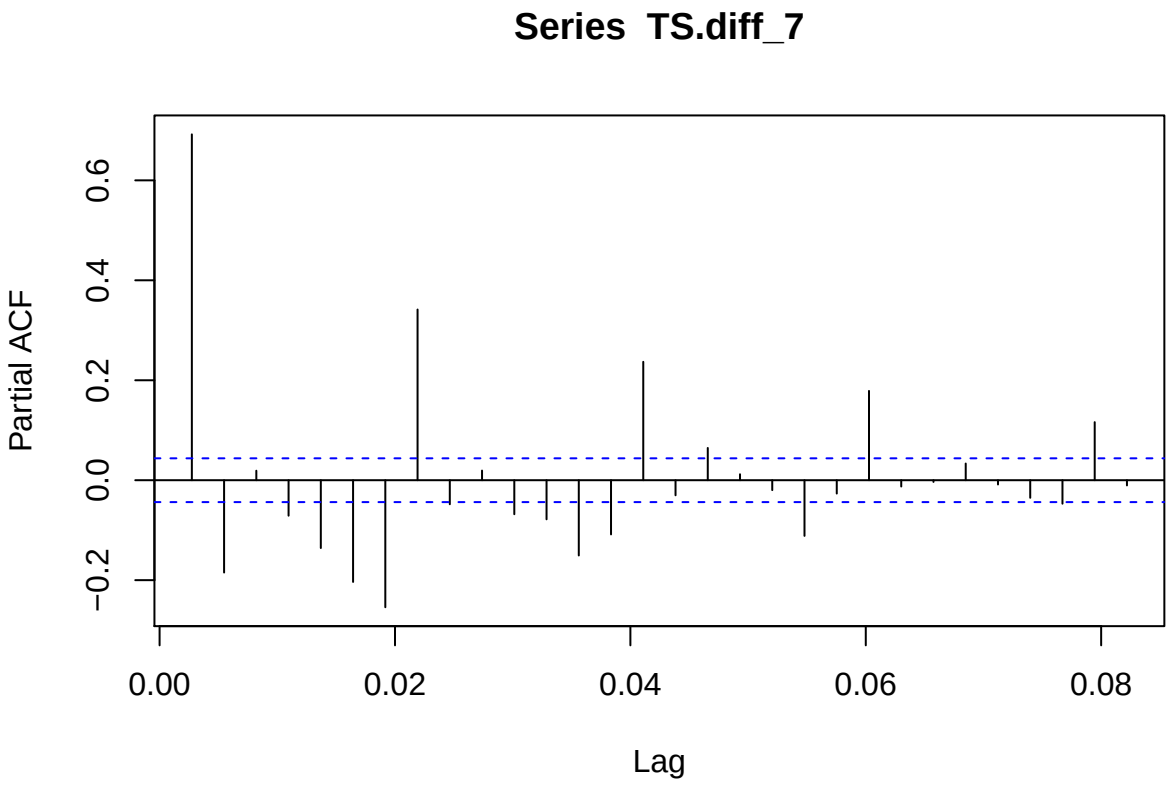



```
ts.plot(TS.diff_7, main="Weekly Differenced Time Series")
```



We now plot the pacf for the weekly time series

```
# Weekly Differenced Time Series
pacf(TS.diff_7, lag.max=30)
```



3) Model Building

A number of traditional time series, Machine Learning, Deep Learning models were used for our forecasting problem. These have been briefly described below:-

1) Traditional Time Series models

AR Model: In a multiple regression model, we forecast the variable of interest using a linear combination of predictors. In an autoregression model, we forecast the variable of interest using a linear combination of *past values of the variable*. The term autoregression indicates that it is a regression of the variable against itself.

Thus, an autoregressive model of order p can be written as

$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + \epsilon_t$ where ϵ_t is white noise.

This is like a multiple regression but with lagged values of
y_t as predictors. We refer to this as an $AR(p)$ model, an autoregressive model of order p. However the use of this model is only restricted to stationary data.

MA model: Rather than using past values of the forecast variable in a regression, a moving average model uses past forecast errors in a regression-like model.

$y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + ... + \theta_q \epsilon_{t-q}$ where ϵ_t is white noise. We refer to this model as an $MA(q)$ model, a moving average model of order q.Of course, we do not observe the values of ϵ_t, so it is not really a regression in the usual sense.

Notice that each value of y_t can be thought of as a weighted moving average of the past few forecast errors.

ARIMA model: If we combine differencing with autoregression and a moving average model, we obtain a non-seasonal ARIMA model. ARIMA is an acronym for AutoRegressive Integrated Moving Average. The full model can be written as

$y_t^{'} = c + \phi_1 y_{t-1}^{'} + \phi_2 y_{t-2}^{'} + ... + \phi_p y_{t-p}^{'} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + ... + \theta_q \epsilon_{t-q} + \epsilon_t$

Here $y_t^{'}$ is a differenced time series (it may have been differenced more than once). The "predictors" on the right hand side include both lagged values of y_t and lagged errors. We call this an $ARIMA(p, d, q)$ model, where

- p = order of the autoregressive part
- d = degree of first differencing involved
- q = order of the moving average part

The same stationarity conditions that are used for autoregressive and moving average models also apply to an ARIMA model.

The *auto.arima()* method from the *forecast* package was used to automatically select the best choice of hyperparameters and determine the best performing AR/MA or ARIMA model.

ARCH & GARCH Autoregressive Conditional Heteroskedasticity, or ARCH, is a method that explicitly models the change in variance over time in a time series. Specifically, an ARCH method models the variance at a time step as a function of the residual errors from a mean process (e.g. a zero mean). A lag parameter must be specified to define the number of prior residual errors to include in the model. A generally accepted notation for an ARCH model is to specify the ARCH() function with the q parameter $ARCH(q)$ where q denotes the number of lag squared residual errors to include in the ARCH model. The approach expects the series is stationary, other than the change in variance, meaning it does not have a trend or seasonal component. An ARCH model is used to predict the variance at future time steps.

For $ARCH(m)$ process: $Var(y_t|y_{t-1}, ..., y_{t-m}) = \sigma^2 = \alpha_0 + \alpha_1 y_{t-1}^2 + ... + \alpha_m y_{t-m}^2$

On a similar note, Generalized Autoregressive Conditional Heteroskedasticity, or GARCH, is an extension of the ARCH model that incorporates a moving average component together with the autoregressive component. Specifically, the model includes lag variance terms (e.g. the observations if modeling the white noise residual errors of another process), together with lag residual errors from a mean process.

The introduction of a moving average component allows the model to both model the conditional change in variance over time as well as changes in the time-dependent variance. Examples include conditional increases and decreases in variance. As such, the model introduces a new parameter "p" that describes the number of lag variance terms:

- p: The number of lag variances to include in the GARCH model.
- q: The number of lag residual errors to include in the GARCH model.

For $GARCH(1, 1)$ process:$\sigma^2 = \alpha_0 + \alpha_1 y_{t-1}^2 + \beta_1 \sigma_{t-1}^2$

2) Machine Learning models

We make use of the *modeltime* package in R to build Machine Learning models.

Lasso & Ridge Regression: Implemented using the glmnet package in R.

Ridge Regression: In Ridge regression, we add a penalty term which is equal to the square of the coefficient. The L2 term is equal to the square of the magnitude of the coefficients. We also add a coefficient λ to control that penalty term. In this case if λ is zero then the equation is the basic OLS else if $\lambda > 0$ then it will add a constraint to the coefficient. As we increase the value of λ this constraint causes the value of the coefficient to tend towards zero. This leads to both low variance (as some coefficient leads to negligible effect on prediction) and low bias (minimization of coefficient reduce the dependency of prediction on a particular variable).

$L_{ridge} = argmin_{\beta}(||Y - \beta * X||^2 + \lambda * ||\beta^2||)$

Lasso Regression : Lasso regression stands for Least Absolute Shrinkage and Selection Operator. It adds penalty term to the cost function. This term is the absolute sum of the coefficients. As the value of coefficients increases from 0 this term penalizes, cause model, to decrease the value of coefficients in order to reduce loss. The difference between ridge and lasso regression is that it tends to make coefficients to absolute zero as compared to Ridge which never sets the value of coefficient to absolute zero.

$L_{lasso} = argmin_{\beta}(||Y - \beta * X||^2 + \lambda * ||\beta||)$

Elastic Net : Sometimes, the lasso regression can cause a small bias in the model where the prediction is too dependent upon a particular variable. In these cases, elastic Net is proved to better it combines the regularization of both lasso and Ridge.

Random Forests The random forest method can build prediction models using random forest regression trees, which are usually unpruned to give strong predictions. The bootstrap sampling method is used on the regression trees, which should not be pruned. Optimal nodes are sampled from the total nodes in the tree to form the optimal splitting feature.

The random sampling technique used in the selection of the optimal splitting feature lowers the correlation and hence, the variance of the regression trees. It improves the predictive capability of distinct trees in the forest. The sampling using bootstrap also increases independence among individual trees.

Gradient Boosted Trees (XGBoost): XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. By combining many weak learners to form a strong learner, thus offering highly accurate predictions. XGBoost improves upon the base GBM framework through systems optimization and algorithmic enhancements such as parallel processing, tree-pruning, regularization etc.

3) Deep Learning models

Unlike the feed-forward neural networks, Recurrent Neural Networks (RNNs) contain hidden states which are distributed across time. This allows them to efficiently store a lot of information about the past. As with a regular feed-forward neural network, the non-linear dynamics introduced by the nodes allows them to capture complicated time series dynamics.

RNNs fall into the category of a dynamic neural network, wherein the output depends on the current input to the network, and the previous inputs, out- puts, and/or hidden states of the network.

An RNN basically takes the output of each hidden or output layer, and feeds it back to itself (via the delay node) as an additional input. The delay unit allows information to be passed from one time step to the next.

The delay unit enables the network to have short-term memory. This is because it stores the hidden layer activation values and/or output values of the previous time step. It releases these values back into the network at the subsequent time step. In other words, the RNN has a "memory" which captures information about what has been calculated by the hidden units at an earlier time step. Time-series data contain patterns ordered by time. Information about the underlying data generating mechanism is contained in these patterns.

RNNs take advantage of this ordering because the delay units exhibit persistence. It is this "short term" memory feature that allows an RNN to learn and generalize across sequences of inputs.

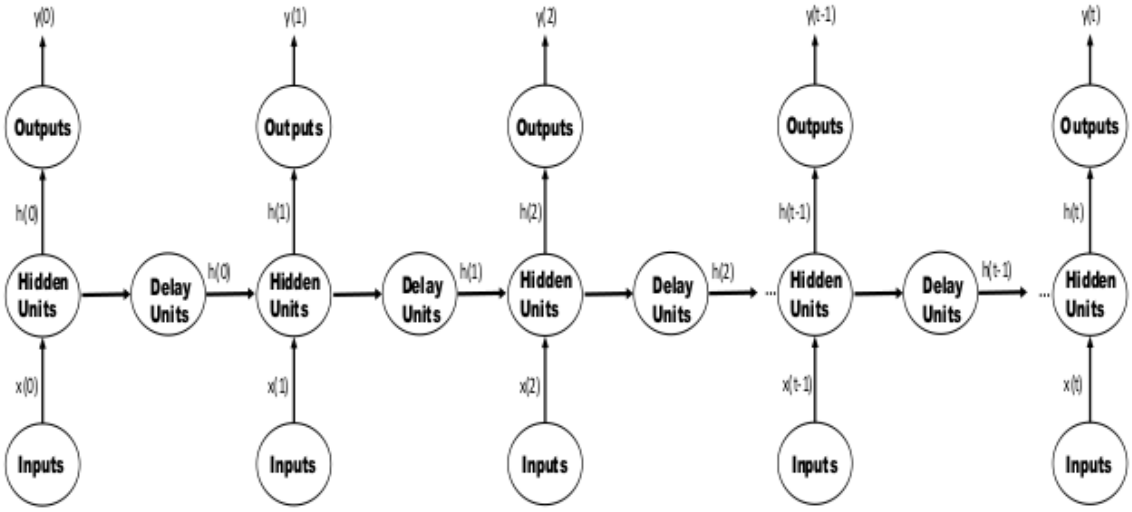


Figure 4: Unfolded Structure of a RNN

The two kinds of recurrent neural networks used by us, are described below

Elman Recurrent Neural Networks Elman neural networks were initially designed to learn sequential or time-varying patterns. They are composed of an input layer, a context layer (also called a recurrent or delay layer), a hidden layer, and an output layer.

Each layer contains one or more neurons which propagate information from one layer to another by computing a nonlinear function of their weighted sum of inputs. In an Elman neural network, the number of neurons in the context layer is equal to the number of neurons in the hidden layer. In addition, the context layer neurons are fully connected to all the neurons in the hidden layer. Similar to a regular feedforward neural network, the strength of all connections between neurons is determined by a weight. Initially, all weight values are chosen randomly and are optimized during training.

Jordan Recurrent Neural Networks A Jordan neural network is a single hidden layer feed-forward neural network. It is similar to the Elman neural network. The only difference is that the context (delay) neurons are fed from the output layer instead of the hidden layer, see Figure below. It therefore "remembers" the output from the previous time-step.

4) Other models

Prophet: Released by Facebook research recently, Prophet is a procedure for forecasting univariate time series data automatically based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects.

It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

The procedure makes use of a decomposable time series model with three main model components: trend, seasonality, and holidays.

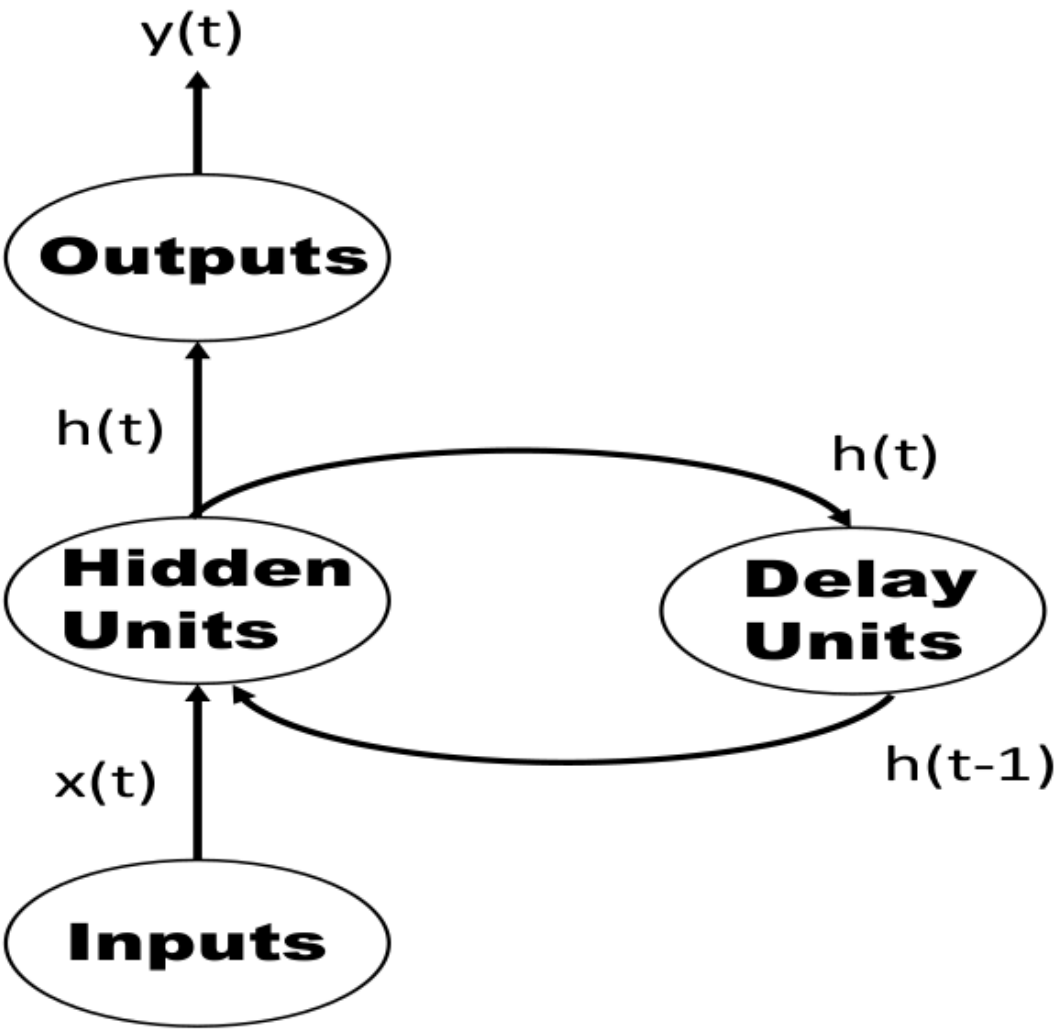


Figure 5: Elman RNN

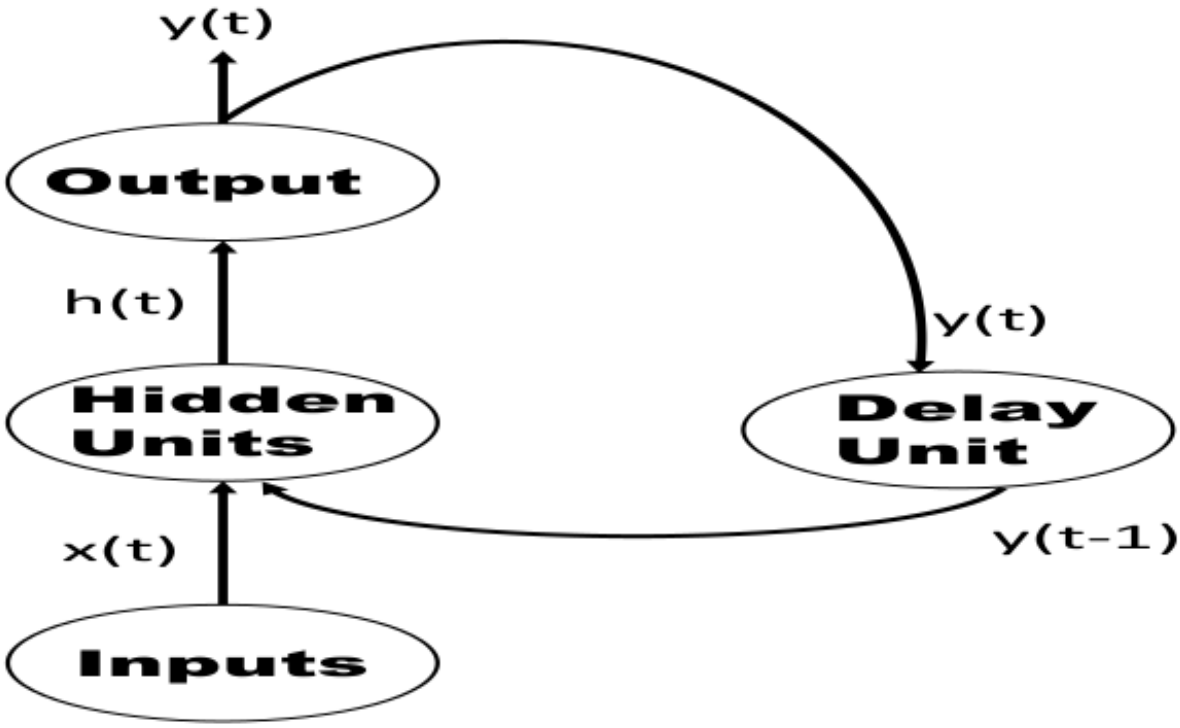


Figure 6: Jordan RNN

Similar to a generalized additive model (GAM), with time as a regressor, Prophet fits several linear and non-linear functions of time as components. In its simplest form;

$y(t) = g(t) + s(t) + h(t) + e(t)$

where: - $g(t)$:trend models non-periodic changes (i.e. growth over time) - $s(t)$:seasonality presents periodic changes (i.e. weekly, monthly, yearly) - $h(t)$:ties in effects of holidays (on potentially irregular schedules 1 day(s)) - $e(t)$:covers idiosyncratic changes not accommodated by the model

Prophet is essentially *"framing the forecasting problem as a curve-fitting exercise"* rather than looking explicitly at the time based dependence of each observation

NOTE: Modeling seasonality as an additive component is the same approach taken by exponential smoothing. GAM formulation has the advantage that it decomposes easily and accommodates new components as necessary, for instance when a new source of seasonality is identified.

Prophet & XGBoost Ensemble: Ensembling is a technique using which predictions from several models can be combined together. So in this model we fit a linear model over the forecasts returned by the two models described earlier.

Results & Discussions

Now in the code given below, we evaluate the performance of AUTO-ARIMA models on *daily,weekly,monthly* scales.

```
# 7-day forecast 19.83% MAPE using Auto Arima
# 1-month forecast 20% MAPE using Auto Arima
```

```
training_1day = as.ts(df_ts[1:2008])
training_1week = as.ts(df_ts[1:2002])
training_1month = as.ts(df_ts[1:1979])
test_1day = df_ts[2009]# 1 day error = 2.6%
test_1week = df_ts[2003:2009]# 7 day error = 19.63%
test_1month = df_ts[1980:2009]# 1 month error = 22%
## use auto.arima to choose ARIMA hyperparams
library(forecast)
fit <- auto.arima(training_1month)
summary(fit)
```

```
## Series: training_1month
## ARIMA(4,1,4)
##
## Coefficients:
##          ar1     ar2     ar3      ar4      ma1      ma2      ma3     ma4
##       0.8019  0.0601  0.2804  -0.2880  -0.9238  -0.2615  -0.1712  0.4085
## s.e.  0.2268  0.2695  0.1783   0.1049   0.2237   0.2896   0.1955  0.1363
##
## sigma^2 estimated as 2567:  log likelihood=-10567
## AIC=21152   AICc=21152.09   BIC=21202.31
##
## Training set error measures:
##                      ME     RMSE      MAE       MPE     MAPE      MASE
## Training set -0.402949 50.54795 37.22287 -4.031646 16.47963 0.9580475
##                      ACF1
## Training set 0.001911221
```

```
## forecast for future time points
predictions <- forecast(fit, h = 30)
y_pred = predictions$mean
library(MLmetrics)
MAPE(y_pred,test_1month)
```

```
## [1] 0.2298436
```

```
library(caret)
R2(y_pred,test_1month)
```

```
## [1] 0.001792622
```

```
MAE(y_pred,test_1month)
```

```
## [1] 26.22967
```

While constructing Auto-ARIMA models, the following accuracies were obtained.

Model	$MAPE$	R^2
Auto-Arima_1_day	2.6%	N.A.
Auto-Arima_1_week	19.83%	0.23
Auto-Arima_1_month	23%	0.01

We now evaluate the performance of the machine learning models built using *modeltime* package

```
library(readr)
AQI_univar <- read_csv("AQI_univar.csv", col_types = cols(X1 = col_skip(), Date = col_date(format = "%Y
df = AQI_univar
# Load required packages
library(tidymodels)
library(modeltime)
library(timetk)
library(lubridate)
library(tidyverse)

# Select columns and set appropriate names
aqi_tbl <- df %>%
  select(Date, AQI) %>%
  set_names(c("Date", "AQI"))


# Split Data into train and test set.
# We can test prediction accuracies while forecasting 1 day, 1 week and 1 month in future respectively
splits <- aqi_tbl %>%
  time_series_split(assess = "1 week", cumulative = TRUE)

# Plot train-test split
splits %>%
  tk_time_series_cv_plan() %>%
  plot_time_series_cv_plan(Date, AQI, .interactive = FALSE)
```

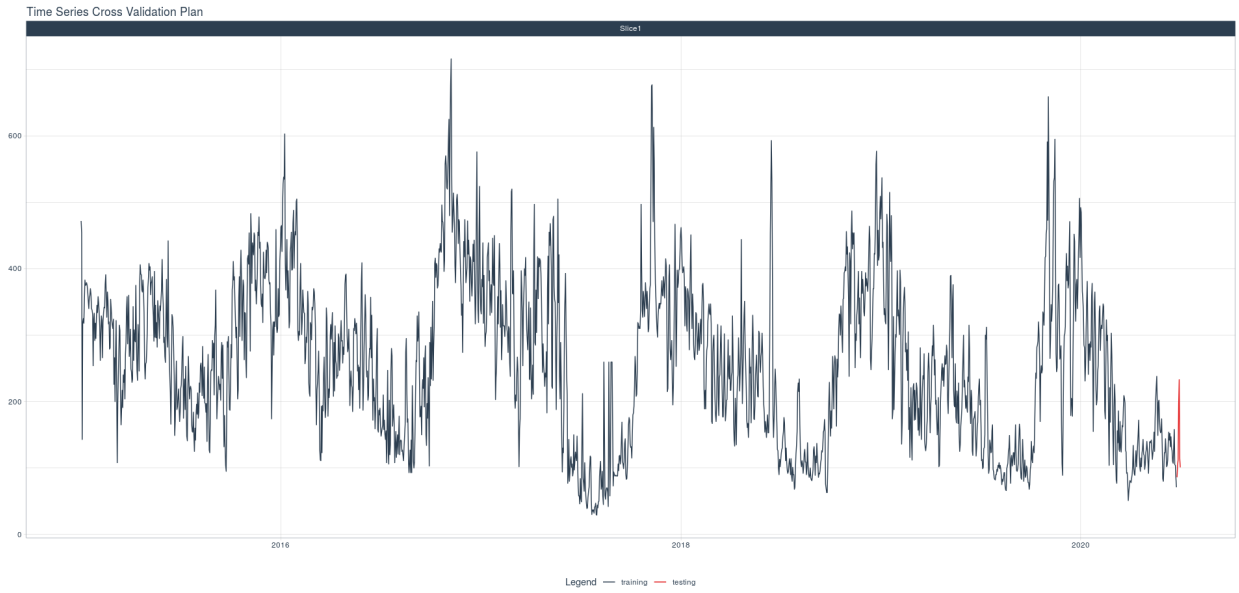


Figure 7: Train-Test Split

Some time series models such as prophet and exponential trend smoothing (ETS) are coded up below.

```r
# Modelling
# Auto-arima
model_fit_arima <- arima_reg() %>%
  set_engine("auto_arima") %>%
  fit(AQI ~ Date, training(splits))

# Prophet
model_fit_prophet <- prophet_reg(seasonality_weekly = TRUE) %>%
  set_engine("prophet") %>%
  fit(AQI ~ Date, training(splits))
model_fit_prophet

# Model : ets ----
model_fit_ets <- exp_smoothing() %>%
  set_engine(engine = "ets") %>%
  fit(AQI ~ Date, training(splits))
#model_fit_ets
```

We now manufacture new features which will serve as inputs to our machine learning models.

```r
## Engineer new features for Machine Learning Models
recipe_spec <- recipe(AQI ~ Date, training(splits)) %>%
  step_timeseries_signature(Date) %>%
  step_rm(contains("am.pm"), contains("hour"), contains("minute"), #Remove time features as they're abs
          contains("second"), contains("xts")) %>%
  step_fourier(Date, period = 365, K = 5) %>%
  step_dummy(all_nominal())

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

```
> colnames(juice(prep(recipe_spec)))
 [1] "Date"              "AQI"               "Date_index.num"    "Date_year"         "Date_year.iso"     "Date_half"
 [7] "Date_quarter"      "Date_month"        "Date_day"          "Date_wday"         "Date_mday"         "Date_qday"
[13] "Date_yday"         "Date_mweek"        "Date_week"         "Date_week.iso"     "Date_week2"        "Date_week3"
[19] "Date_week4"        "Date_mday7"        "Date_sin365_K1"    "Date_cos365_K1"    "Date_sin365_K2"    "Date_cos365_K2"
[25] "Date_sin365_K3"    "Date_cos365_K3"    "Date_sin365_K4"    "Date_cos365_K4"    "Date_sin365_K5"    "Date_cos365_K5"
[31] "Date_month.lbl_01" "Date_month.lbl_02" "Date_month.lbl_03" "Date_month.lbl_04" "Date_month.lbl_05" "Date_month.lbl_06"
[37] "Date_month.lbl_07" "Date_month.lbl_08" "Date_month.lbl_09" "Date_month.lbl_10" "Date_month.lbl_11" "Date_wday.lbl_1"
[43] "Date_wday.lbl_2"   "Date_wday.lbl_3"   "Date_wday.lbl_4"   "Date_wday.lbl_5"   "Date_wday.lbl_6"
```

Figure 8: Train-Test Split

Now since all these new features have automatically been engineered, we code up the Machine Learning models below.

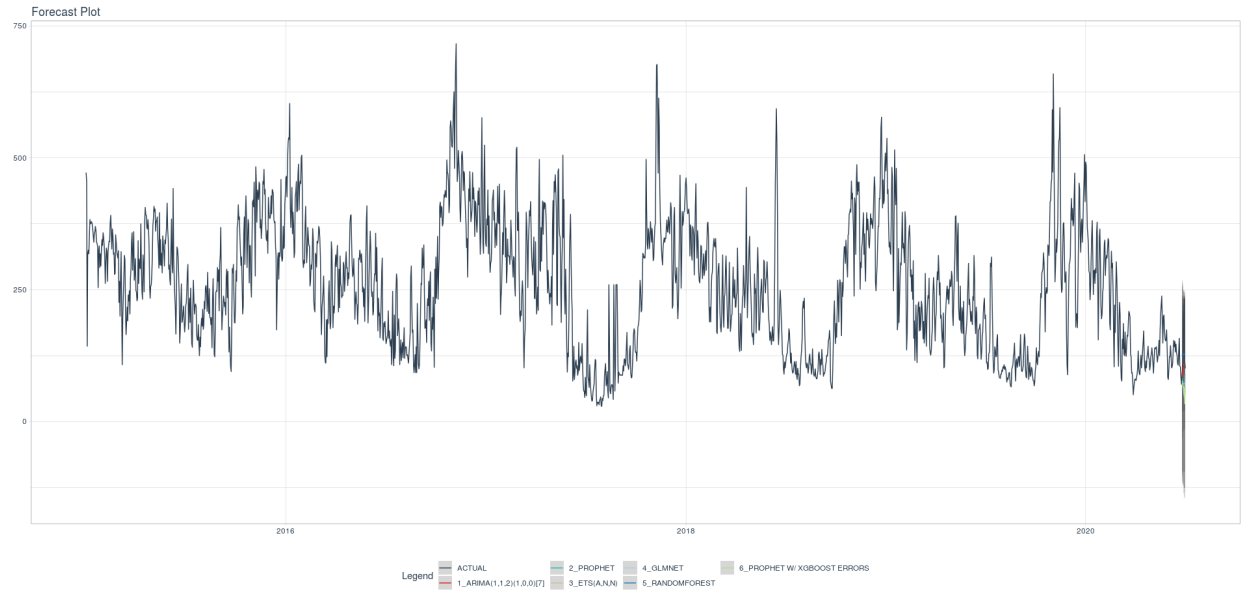
```r
# Elastic Net: Lasso+ Ridge Regularized Regression
model_spec_glmnet <- linear_reg(penalty = 0.01, mixture = 0.5) %>%
  set_engine("glmnet")
workflow_fit_glmnet <- workflow() %>%
  add_model(model_spec_glmnet) %>%
  add_recipe(recipe_spec %>% step_rm(Date)) %>%
  fit(training(splits))
```

```
# Random Forest: Bagged Collection of Decision Trees

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))

# Prophet Boost = Prophet + XGBoost ensemble
model_spec_prophet_boost <- prophet_boost(seasonality_weekly = TRUE) %>%
  set_engine("prophet_xgboost")

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

workflow_fit_prophet_boost
```

The models are now evaluated on unseen data.

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

model_table


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

calibration_table

# Forecast
calibration_table %>%
  modeltime_forecast(actual_data = aqi_tbl) %>%
  plot_modeltime_forecast(.interactive = FALSE)

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

Training Fit for all the models.



Model Accuracies for 1-week forward forecasts

Accuracy Table

.model_id	.model_desc	.type	mae	mape	mase	smape	rmse	rsq
1	ARIMA(1,1,2)(1,0,0)[7]	Test	35.59	18.61	0.77	23.61	59.24	0.24
2	PROPHET	Test	68.19	42.84	1.47	60.00	89.40	0.26
3	ETS(A,N,N)	Test	63.44	40.35	1.36	53.86	81.93	NA
4	GLMNET	Test	69.96	43.04	1.50	62.25	94.32	0.48
5	RANDOMFOREST	Test	42.18	28.95	0.91	29.70	52.42	0.00
6	PROPHET W/ XGBOOST ERRORS	Test	72.74	47.62	1.56	67.54	90.54	0.00

Model Accuracies for 1-month forward forecasts

```
## Refit and forecast future AQI
calibration_table %>%
  # Remove ARIMA model with low accuracy
  #filter(.model_id != 2) %>%

  # Refit and Forecast Forward 1 month in future
  modeltime_refit(aqi_tbl) %>%
  modeltime_forecast(h = "1 months", actual_data = aqi_tbl) %>%
  plot_modeltime_forecast(.interactive = FALSE)
```

Deep Learning Models

Accuracy Table

.model_id	.model_desc	.type	mae	mape	mase	smape	rmse	rsq
1	ARIMA(1,1,2)(1,0,0)[7]	Test	24.43	21.20	1.10	19.04	32.46	0.02
2	PROPHET	Test	30.71	22.61	1.38	25.72	47.60	0.00
3	ETS(A,N,N)	Test	45.24	32.87	2.03	40.97	54.80	NA
4	GLMNET	Test	39.93	30.38	1.80	30.96	57.05	0.00
5	RANDOMFOREST	Test	39.29	35.78	1.77	29.06	44.95	0.00
6	PROPHET W/ XGBOOST ERRORS	Test	52.44	39.93	2.36	60.69	75.49	0.00

Figure 9: Model Accuracies for 1-month forward forecasts

Elman Recurrent Neural Networks: In the code block given below, we implement Elman & Jordan RNNs

```
library(tseries)
library(quantmod)
y <-as.ts(df_ts)
y <-log(y)# Normalize the data to stabilize values
#y <- as.ts(scale(y)) # Scale the dataset before plugging inside Neural Network
y <-as.zoo(y)

###### Model-1 (Including lags of past 12 months) ######

x1=Lag (y , k = 1)
x2=Lag (y , k = 2)
x3=Lag (y , k = 3)
x4=Lag (y , k = 4)
x5=Lag (y , k = 5)
x6=Lag (y , k = 6)
x7=Lag (y , k = 7)
dataset_elman <- cbind( x1 , x2 , x3,
                 x4 , x5 , x6 ,
                 x7 )
dataset_elman <- cbind (y,dataset_elman)
#head(round(dataset_elman,2),14)
dataset_elman <- dataset_elman[-(1:7),]
head(round(dataset_elman,2),6)# First 7 values contain NA so they need to be removed
```

```
##             y Lag.1 Lag.2 Lag.3 Lag.4 Lag.5 Lag.6 Lag.7
## 2015(8)  5.95  5.87  5.76  5.78  5.77  4.96  6.12  6.16
## 2015(9)  5.93  5.95  5.87  5.76  5.78  5.77  4.96  6.12
## 2015(10) 5.93  5.93  5.95  5.87  5.76  5.78  5.77  4.96
## 2015(11) 5.94  5.93  5.93  5.95  5.87  5.76  5.78  5.77
## 2015(12) 5.93  5.94  5.93  5.93  5.95  5.87  5.76  5.78
## 2015(13) 5.90  5.93  5.94  5.93  5.93  5.95  5.87  5.76
```

```
n = nrow(dataset_elman)
n_train<-1972
train = 1:1972
test = 1973:2002
#train <- sample(1: n , n_train , FALSE )
#test = -train
inputs <- dataset_elman[,2:8]
outputs <- dataset_elman[,1]

library(RSNNS)
fit1 <- elman(inputs[train],outputs[train],size =c(10),
              learnFuncParams =c(0.01),maxit =1000)
fit2 <- jordan(inputs[train], outputs[train],size =c(10),learnFuncParams =c (0.01),maxit=1000)
#par(mfrow =c (1,1) )
#plotIterativeError(fit1)
#summary(fit1)
pred <- predict(fit1, inputs[test])
library(caret)
library(MLmetrics)
MAPE(pred,outputs[test])
```

```
## [1] 0.0334298
```

```
R2(outputs[test],pred)
```

```
##           [,1]
## [1,] 0.3365026
```

The results obtained using RNNs look quite promising and are described in detail below. Each of the networks given below is a single layered neural network having 10 neurons in the hidden layer.

Model	$MAPE$	R^2
ElmanRNN_1_day	2.07%	N.A.
ElmanRNN_1_week	4.20%	0.40
ElmanRNN__1_month	3.22%	0.37
JordanRNN_1_day	1.13%	N.A.
JordanRNN_1_week	3.91%	0.48
JordanRNN_1_month	3.53%	0.39

Forecasts for the next month are plotted below.

Conclusion

After using 3 different categories of models, our findings show that currently RNNs forecast the monthly AQI with minimum error. (3%-4%) We have tackled this problem by taking a univariate approach. We believe that once we look at it from a multivariate angle, with added features relating to climatic conditions, even the core Time Series model like ARIMA will perform well.

Through this project, we have not just forecasted the AQI but also analyzed the Time Series data beforehand in order to understand the data better. We additionally observed that when we only took last year's data

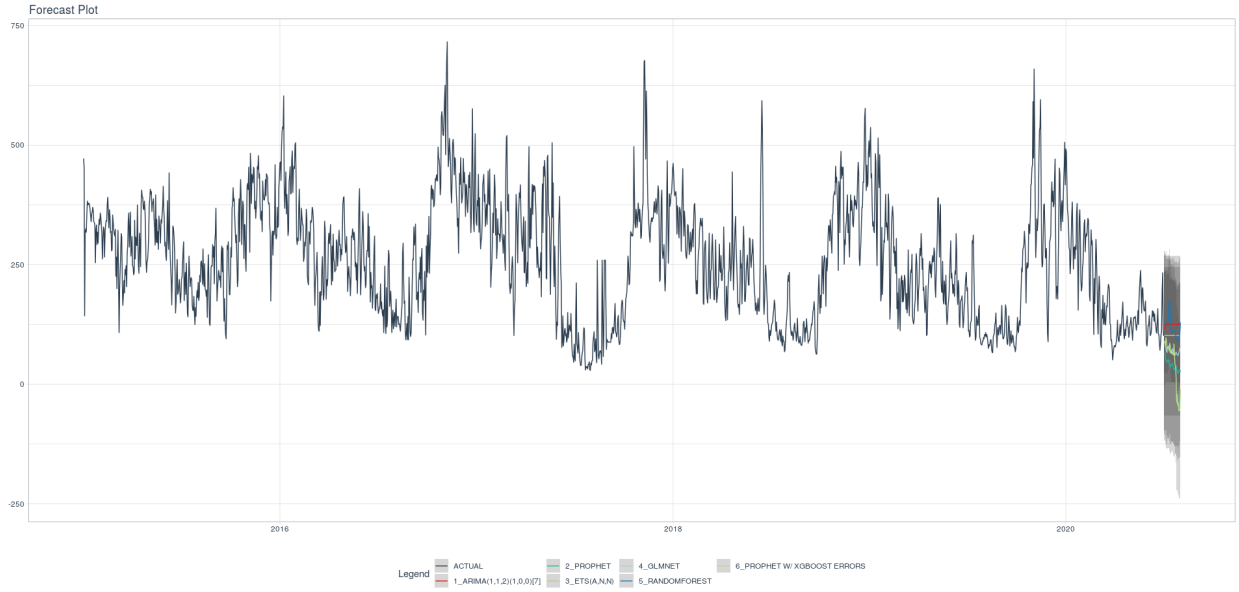


Figure 10: 1-month forward

to forecast a month's AQI, the accuracy of ARIMA increased significantly, by about 11%. This opens the scope of exploring the amount of past data to consider for forecasting, which becomes our future scope.