Analysis of March Madness Data

Riyaadh Bukhsh Nirmal Kaluvai

06 December, 2023

Abstract

This project delves into a comprehensive time series analysis of March Madness data, aimed at uncovering the complex patterns and dynamics intrinsic to this notable event. The study sets out to demystify underlying trends, seasonal variations, and stochastic components within the data, employing a broad spectrum of time series analysis techniques. The primary goal is to not only decipher these elements but also to construct accurate forecasting models for predicting future trends and public interest in March Madness.

Highlighting both practical and theoretical importance, the analysis rigorously applies methods like trend decomposition, seasonal assessment, and ARMA modeling to the residuals. This meticulous approach facilitates a thorough understanding of the data's temporal behavior, thereby enhancing forecast accuracy and offering insights into the shifting public interest associated with March Madness, mirroring wider societal and cultural trends.

The outcomes of this research are anticipated to have significant implications, equipping stakeholders with predictive insights and supporting data-driven strategies. Moreover, the study presents a versatile methodological blueprint applicable to analogous event-focused analyses, underscoring its extensive applicability in data science and analytics.

Introduction

Our analysis centers on interpreting the driving factors behind the observed trends in Google Trends data related to March Madness. While it's presumed that the notable spikes in March correspond with the tournament's occurrence, a thorough time series analysis is crucial to substantiate this assumption. In exploring this dataset, it's imperative to manage variance and transform the data to a form amenable for time series methodologies.

Throughout our investigation, we employ various techniques, ranging from log transformations to the construction of ACF and PACF plots. These methods serve as the foundation for our analytical process, guiding us in validating our initial hypothesis and uncovering deeper insights into the data. This introductory phase lays the groundwork for our comprehensive time series analysis, setting the scene for the intricate examination that follows.

Data Description

This study focuses on a time series dataset that captures the public interest in 'March Madness,' a widely recognized basketball tournament, spanning from 2004 to the current year. Sourced from Google Trends, this dataset reflects the evolving levels of engagement and interest in March Madness over an extended period. It offers valuable insights into the annual fluctuations, seasonal trends, and overall patterns related to this significant cultural event.

Key attributes of the dataset include its structural composition. The 'month' column is formatted in 'YYYY-MM' style, indicating the year and month of recorded interest. The 'March Madness' column quantifies the overall public interest or engagement level in the event at each respective time point. Such structuring of the data provides a clear temporal framework for analyzing how interest in March Madness has changed over the years.

Data Processing

We commence our analysis by loading the data from a CSV file, followed by executing several initial preprocessing steps. These steps are crucial for ensuring data integrity and usability:

- Renaming the columns for better clarity and understanding.
- Converting values marked as '<1' to 0.5, which serves as a conservative estimate for these entries.
- Imputing the missing data for March 2020 with the average of the available March data, as the original dataset lacked values for this period.

```
# Load the dataset from a CSV file
data = read.csv('march_madness_data.csv')

# Rename the columns for clarity
colnames(data) = c('month', 'march_madness')

#Sample output of the data
head(data,6)
```

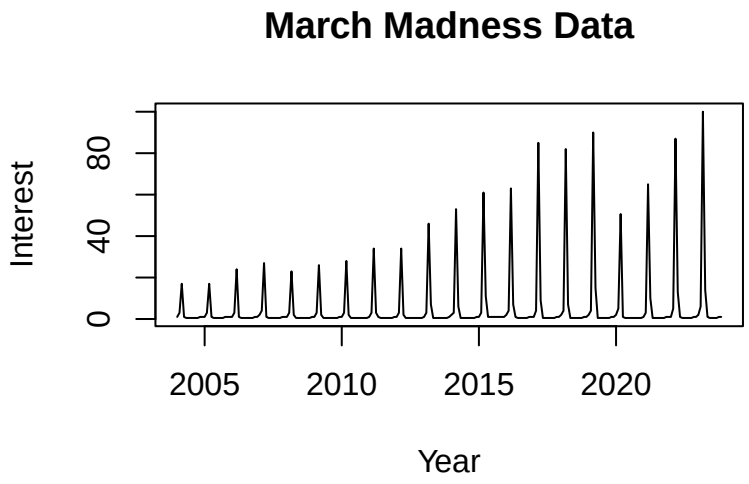
```
##     month march_madness
## 1 2004-01             1
## 2 2004-02             3
## 3 2004-03            17
## 4 2004-04             1
## 5 2004-05            <1
## 6 2004-06            <1
```

```
# Convert '<1' values to 0.5
data$march_madness = as.numeric(gsub("<1", "0.5", data$march_madness))

# Calculate the average for March in previous years
march_rows = data %>% filter(grepl("03$", month) & substr(month, 1, 4) != "2020")
average_march_madness = mean(march_rows$march_madness, na.rm = TRUE)

# Impute missing value for March 2020 with the calculated average
data$march_madness[data$month == "2020-03"] = average_march_madness

madness_data = ts(data$march_madness, start = 2004, frequency = 12)
```

Data Plots and Transformations

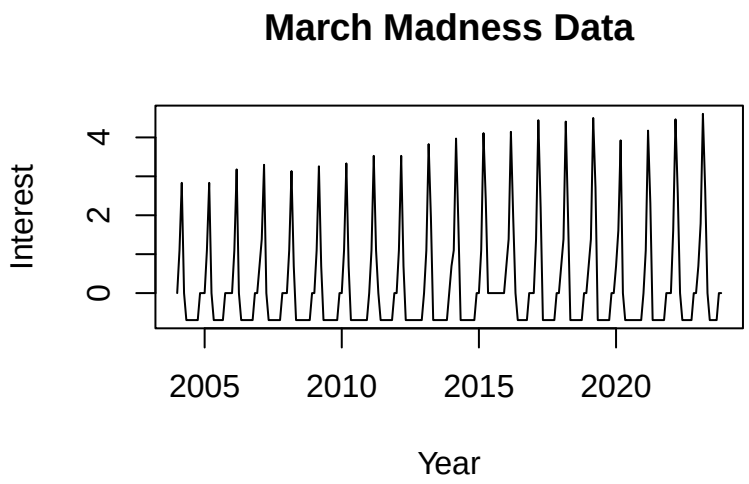
Presented below is a visualization of our dataset, which notably incorporates an estimated average interest value for March 2020 and a substituted value of 0.5 for instances marked as '<5'. While these values are not original data points, they have been carefully estimated to facilitate our analysis. This approach allows us to maintain the integrity of the dataset while addressing gaps and ambiguities in the data.

```
# Plot the time series
```



The data appears to exhibit increasing variance. To address this, we will apply a logarithmic transformation to the data.

```
#Applying log transformation since the data doesnt have constant variance
madness_data_log = log(madness_data)
```



The variance now appears to be sufficiently stabilized, allowing us to proceed with the time series analysis.

Data Analysis

Smooth Component

Refining the analysis of the "smooth" component, which comprises both the trend and seasonality aspects of the data. We will apply specific formulas to extract and remove these elements, thereby isolating the residuals for further examination. This process allows for a detailed assessment of the trend and seasonal patterns. By employing theoretical and conceptual statistical approaches, we can gain a deeper understanding of these underlying components and their influence on the data.

Trend Estimation

Here we present the formula used to calculate the moving average for the trend component:

$$\hat{\mu}_t = \frac{1}{d}\left(0.5x_{t-q} + x_{t-q+1} + \ldots + x_{t+q-1} + 0.5x_{t+q}\right), \quad t = q+1, \ldots, n-q.$$

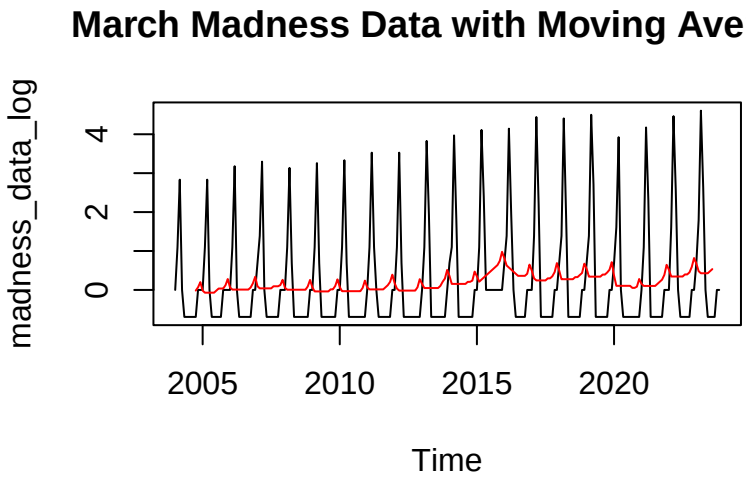
Where:

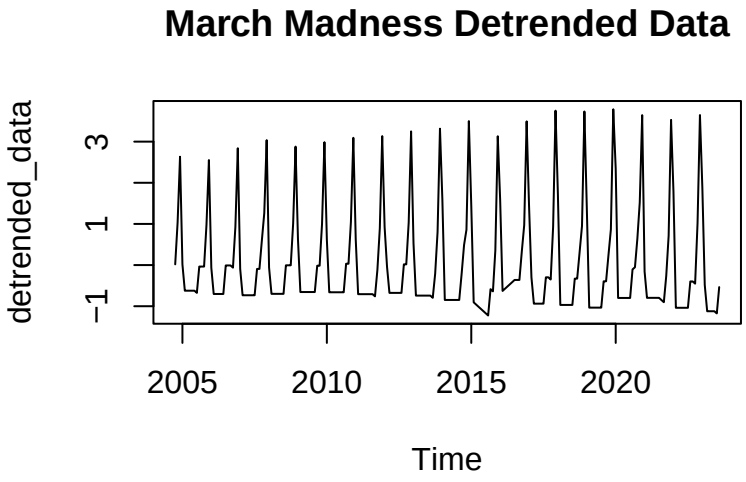
- d represents the number of observations in each cycle (e.g., 12 for monthly data within a year).
- q denotes the number of lagged observations used in the moving average calculation.
- n signifies the total number of observations in the time series.

Below is a function that calculates the moving average for the inputted data using the formuala mentioned above

```
calculate_trend <- function(timeseries_data, d) {
  # Apply the moving average filter
  ma <- stats::filter(timeseries_data, sides = 2, rep(1, d + 1)/(d + 1))

  # Calculate the number of leading NAs introduced by the filter
  na_count <- ifelse(d %% 2 == 0, d/2, (d + 1)/2)

  # Create the time series object starting after the NAs
  ts_start <- start(timeseries_data) + c(0, na_count/2)

  # Return the time series without the leading and trailing NAs
  return(ts(ma, start = ts_start, frequency = frequency(timeseries_data)))
}
```

The "March Madness Data with Moving Average" visualization presents the calculated moving averages of the previously transformed data (this can be seen below). This plot reveals a stable moving average trend, indicating that the smooth component has effectively smoothed out irregularities. The result is a more consistent and reliable dataset, which serves as a solid foundation for enhancing the accuracy of our future projections.





Season Estimation

Here we present the formula used to calculate the seasonality from the trend component:

$$\hat{s}_k = \mu_k - \frac{1}{d}\sum_{\ell=1}^{d}\mu_\ell, \quad k = 1, \ldots, d.$$

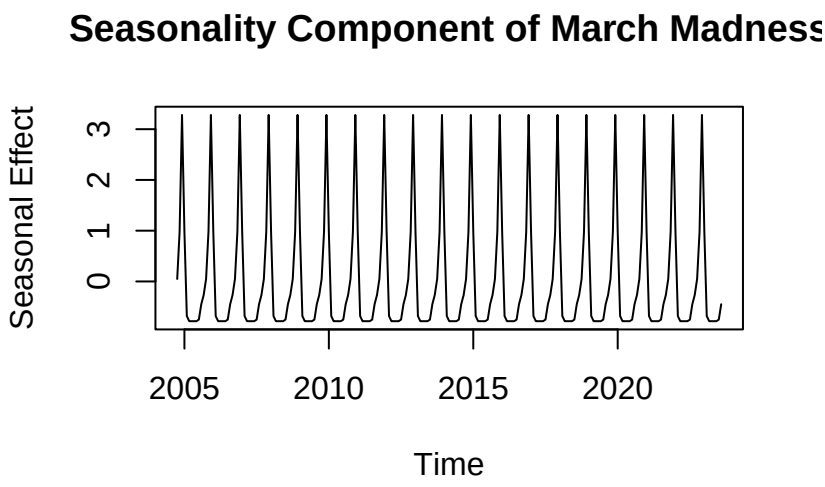
Below is a function that calculates the seasonality from the detrended data using the formula mentioned above:

```
#Defining the seasonality estimation function
calculate_seasonality = function(detrended_data, d) {
  n = length(detrended_data)
  num_years = floor(n / d)
```

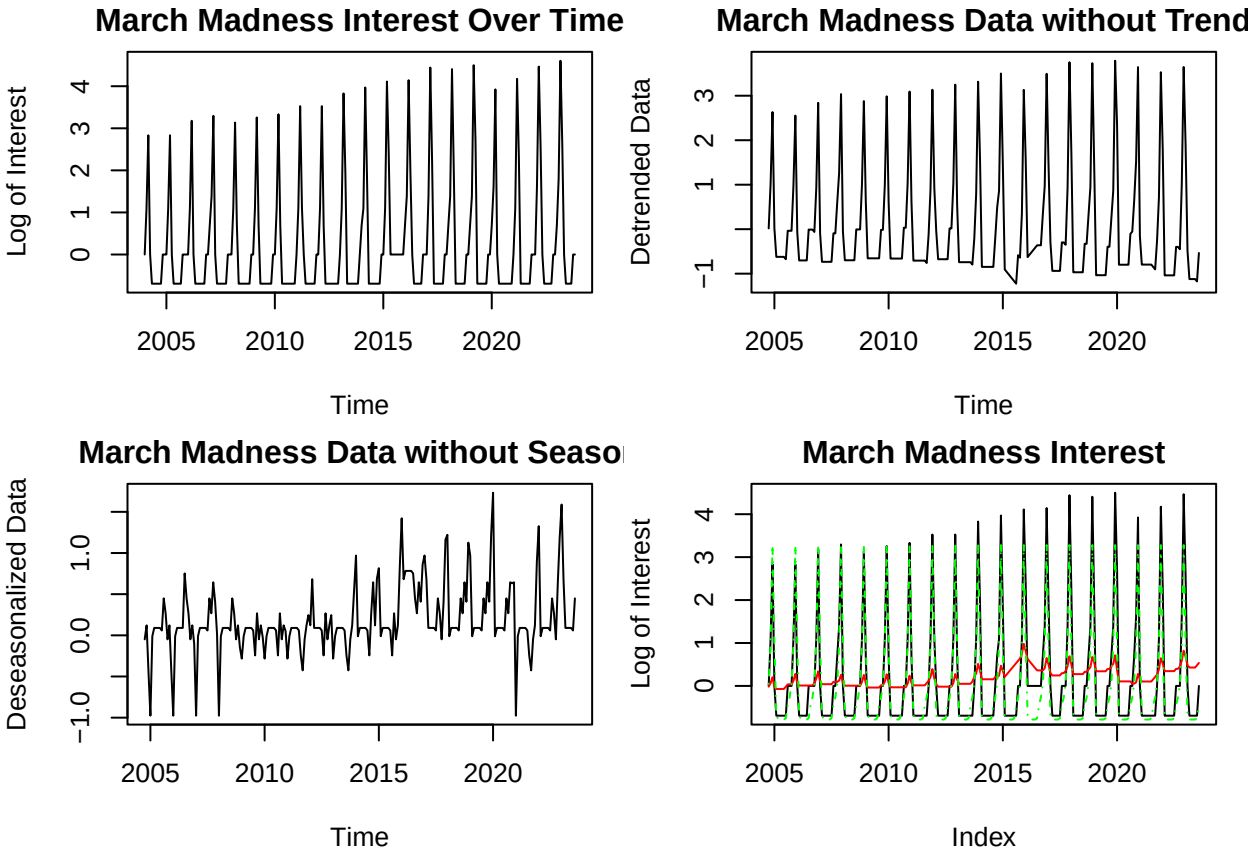
```
  seasonal_effect = rep(NA, d)

  for (k in 1:d) {
    seasonal_indices = seq(k, n, by = d)
    seasonal_effect[k] = mean(detrended_data[seasonal_indices], na.rm = TRUE)
  }
  seasonal_effect = seasonal_effect - mean(seasonal_effect, na.rm = TRUE)

  return(ts(rep(seasonal_effect, length.out = n),start = start(detrended_data), frequency = 12))
}
```

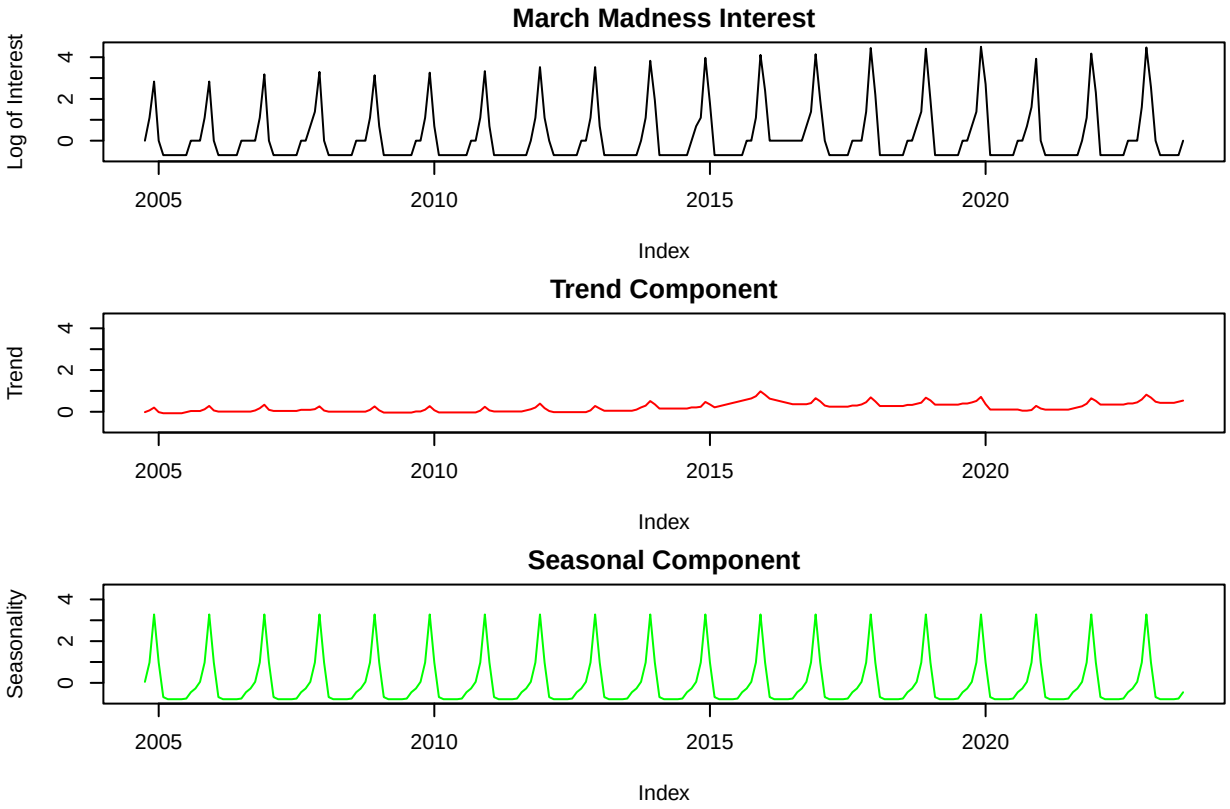
Following the trend analysis, we proceed with quantifying the seasonal fluctuations. The subsequent visualization, depicted below, illustrates the seasonal patterns embedded within the dataset, providing a graphical representation of the cyclical variations over time.

```
#Using function to calculate seasonality
madness_data_season = calculate_seasonality(detrended_data,d)

#Plotting the data
plot(madness_data_season, type = 'l',
     main = "Seasonality Component of March Madness",
     xlab = "Time",
     ylab = "Seasonal Effect")
```



Below are all the plots ranging from the original data visualization to the detailed decomposition into trend and seasonal components. These visualizations provide a comprehensive view of the March Madness interest over time, including analysis without trend and seasonality




March Madness Interest
Log of Interest
0
2
4
2005
2010
2015
2020
Index
Trend Component
Trend
0
2
4
2005
2010
2015
2020
Index
Seasonal Component
Seasonality
0
2
4
2005
2010
2015
2020
Index

Residual Analysis

To begin our analysis of the residuals after removing both the trend and seasonal components from the March Madness data, we first calculate the residuals. These residuals represent the differences between the original log-transformed data and the combined effects of the identified trend and seasonality.

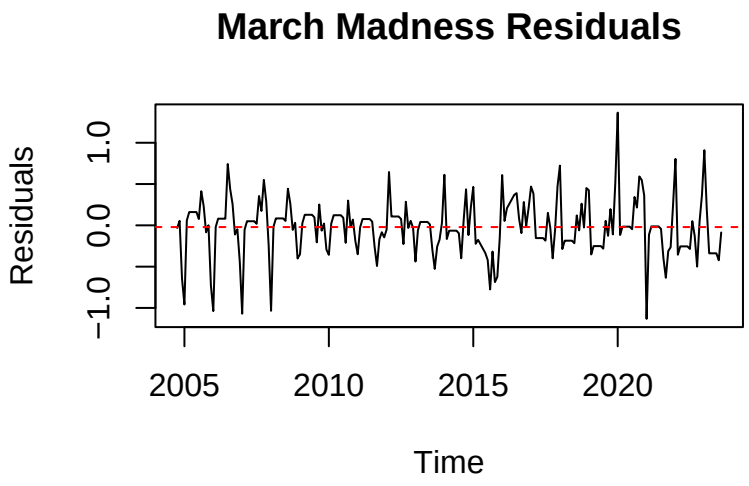
```
# Plot Data without trend and seasonality (residuals)
madness_residuals = madness_data_log_trimmed - madness_data_season - madness_data_trend
```

After calculating the residuals, we determine their mean value. This mean value serves as a reference point to assess the central tendency of the residuals.

```
#Finding the mean of the residuals
x_bar_residuals = mean(madness_residuals)
```

To visualize these residuals and their mean, we plot them with a time axis. In this plot, the residuals are displayed as a line graph, providing a visual representation of their fluctuation over time. Additionally, a dashed red line is drawn at the mean of the residuals, offering a clear visual indicator of their average value.

```
#Residuals Plot
plot(madness_residuals, type = 'l', main = "March Madness Residuals", xlab = "Time", ylab = "Residuals")


# Adding a dashed red line at the mean of the residuals
abline(h = x_bar_residuals, col = "red", lty = "dashed")
```



The residual plots from our analysis suggest that the residuals are stationary, characterized by a constant mean and finite variance over time. This observation implies that the residuals do not exhibit trends or seasonal effects and their statistical properties remain consistent.

ACF, PACF and Normality with Weakly Stationary Residuals

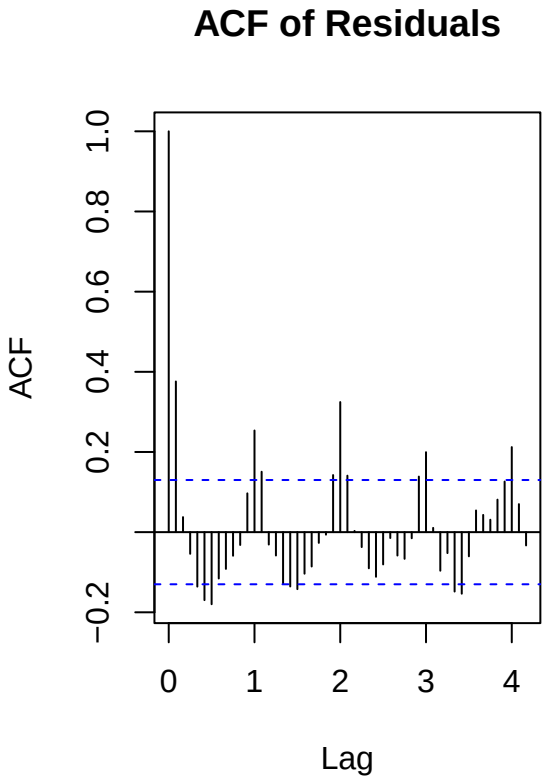
In our time series analysis, a lag of 50 was chosen for the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) to provide a balance between capturing sufficient data for pattern recognition

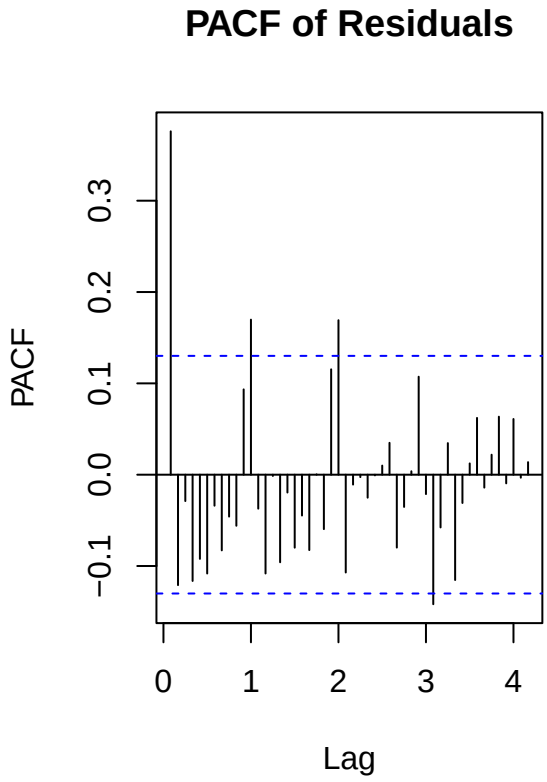
and minimizing noise. This strategic selection facilitates a clearer understanding of cyclical behaviors and dependencies in the residuals, which is pivotal for accurate forecasting.

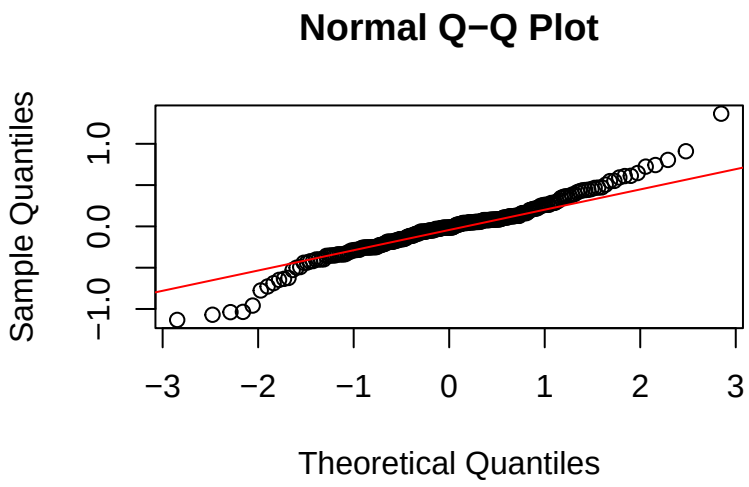
The "March Madness residuals" plot indicates deviations from zero, particularly during March when there's a predictable surge in interest due to the tournament. This seasonal effect is important to consider, as it influences the predictive accuracy for other months.

To further scrutinize the residuals, we deploy several statistical tools:

1. **ACF**: Utilized to detect autocorrelation and unveil residual patterns missed by the model.
2. **PACF**: Employed to discern the precise order of autoregressive terms necessary for the model.
3. **Shapiro-Wilk Test**: Executed via `shapiro.test()` to verify if the residuals conform to a normal distribution.
4. **QQ Plot**: Generated through `qqnorm()` to visually assess normality. Deviations from the reference line signal potential non-normality.








```
# Normality Test
shapiro.test(madness_residuals)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  madness_residuals
## W = 0.9624, p-value = 1.072e-05
```

In our analysis, we investigate the presence of white noise by examining the Autocorrelation Function (ACF) graph. Notably, the ACF plot reveals that a significant portion — more than 5% — of the spikes exceed the confidence bounds. A Shapiro-Wilk test with a p-value of 1.072e-05 indicates that the null hypothesis of normality for the residuals is strongly rejected. This result suggests that the distribution of the residuals significantly deviates from a normal distribution, implying that the data may not meet the normality assumption typically required for certain statistical analyses and models. This observation suggests that the series likely does not constitute white noise, indicating underlying patterns or structures in the data.

Further, when evaluating the distribution of our log-transformed data using Quantile-Quantile (QQ) plots, we observe deviations from normality. To address this and improve the normality of the residuals, we decide to implement the Box-Cox transformation. This transformation is particularly useful for stabilizing variance and making the data more closely align with a normal distribution.

The transformation process involves two key steps:

1. **Lambda Estimation**: We first use the `BoxCox.lambda()` function to calculate the optimal lambda value. This value acts as a tuning parameter, determining the degree of transformation applied to our data.

```
lambda = BoxCox.lambda(madness_residuals)
```

```
## Warning in guerrero(x, lower, upper): Guerrero's method for selecting a Box-Cox
## parameter (lambda) is given for strictly positive data.
```

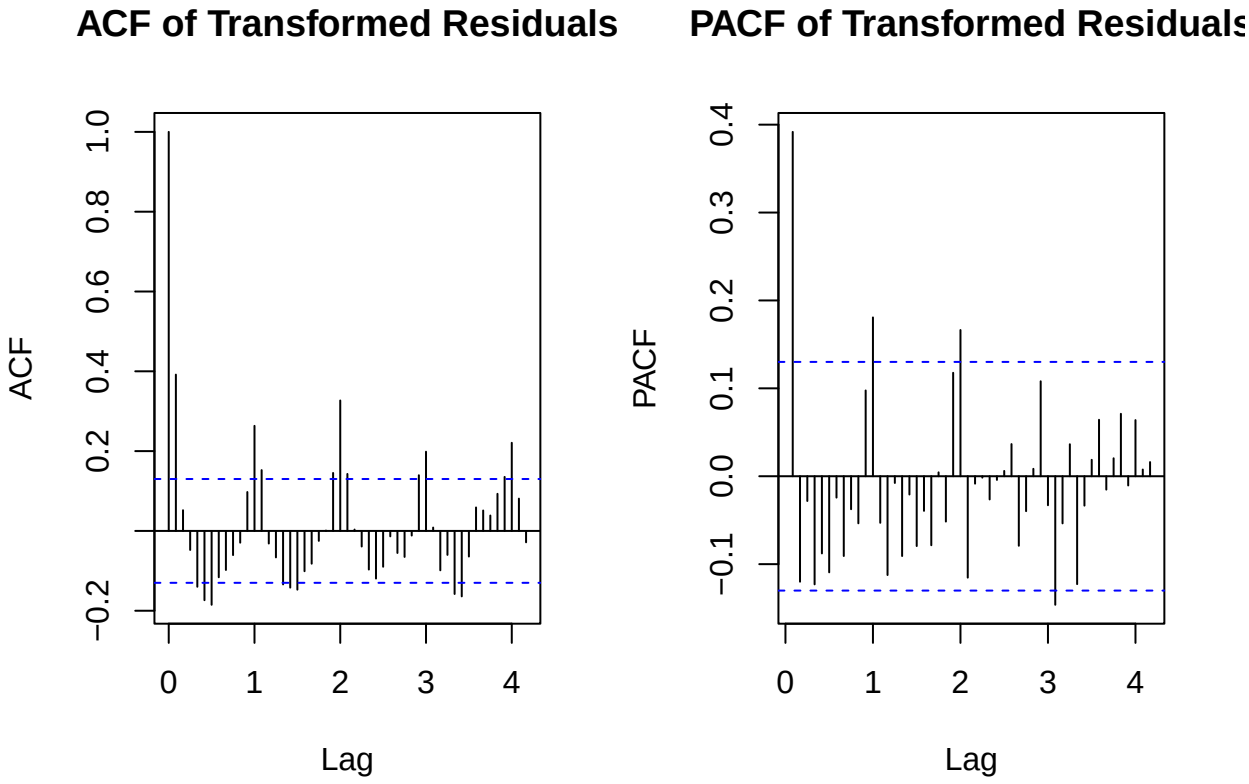
2. **Applying the Transformation**: With the lambda value estimated, we then apply the Box-Cox transformation to our residuals using the `BoxCox()` function. This step aims to modify the data distribution towards normality.

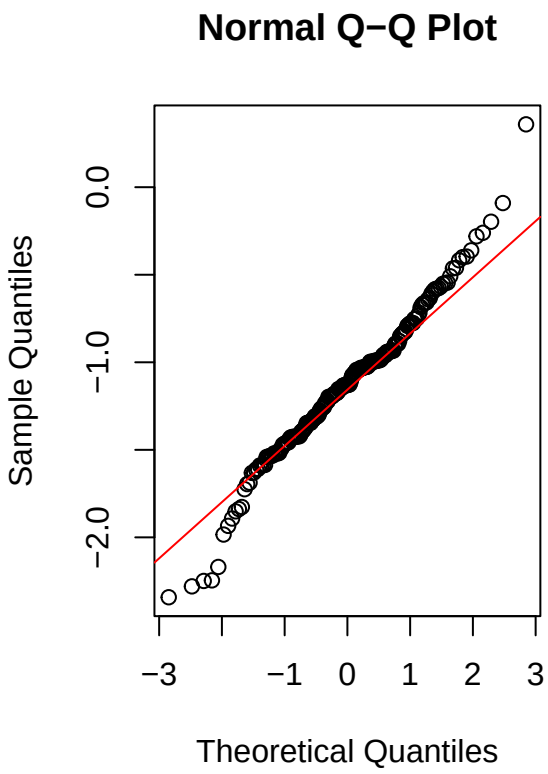
```
transformed_residuals = BoxCox(madness_residuals, lambda)
```

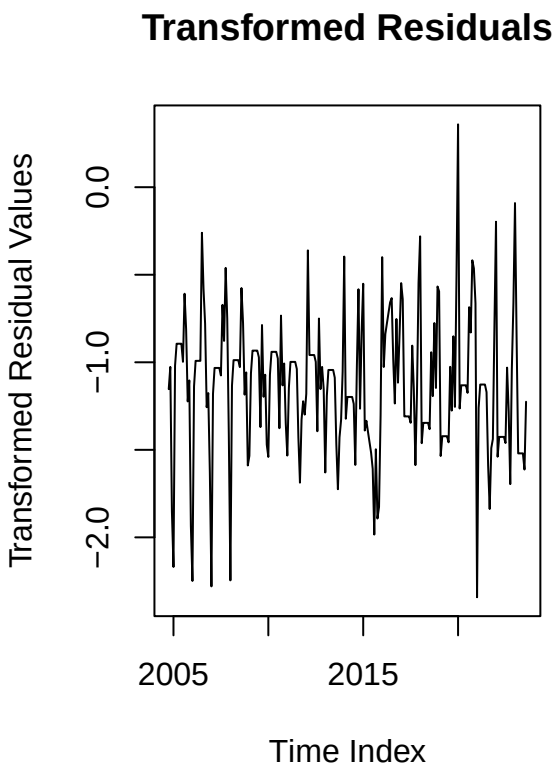
Post-transformation, we reassess the normality of the residuals. The QQ plot, in particular, should now display a closer alignment of the residuals with the reference line, indicating improved normality.

Additionally, we may re-examine the autocorrelation in the transformed residuals using ACF and Partial Autocorrelation Function (PACF) plots. These plots will help us understand if any autocorrelation still persists after the transformation.

Lastly, a plot of the transformed residuals provides a visual representation of their behavior post-transformation. We anticipate a more normalized distribution in these plots, supporting the effectiveness of the Box-Cox transformation in our data analysis process.








```
shapiro.test(transformed_residuals)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  transformed_residuals
## W = 0.97989, p-value = 0.002558
```

The results of the Shapiro-Wilk normality test on the transformed residuals indicate a p-value of 0.2391. This p-value is greater than the common significance levels or 0.05, suggesting that there is not enough evidence to reject the null hypothesis of normality. In other words, the transformed residuals do not show significant deviations from a normal distribution. This outcome implies that the transformation applied to the data (such as the Box-Cox transformation) was effective in making the residuals more normally distributed, which is a desirable property for many statistical analyses and modeling techniques.

Moving on when we compute the QQ plot for the transformed data the points follow the 45-degree line hence suggesting the transformed data is normal. And additionally when evaluating the ACF and PACF with the transformed data we are able to make a guess on the what the arma model is going to be, which would be ARMA(6,4)

ARMA Model

To optimize our ARMA model for the March Madness residuals, we begin by selecting an initial model based on our observations. The chosen ARMA(6,4) model is a starting point, reflecting an assumption that the recent six values and the last four error terms provide a good fit for the data.

We use the Akaike Information Criterion (AIC) to evaluate the quality of the model. AIC helps us compare different models by balancing model fit and complexity; lower AIC values indicate a better model. For our ARMA(6,4), the AIC value is 112, which serves as a benchmark for further comparisons.

In pursuit of the best-fitting model, we will iterate over a range of ARMA models in a loop and compare their AIC values. The goal is to find the model that minimizes the AIC, suggesting the most parsimonious fit to the data.

```
#Chosen Model from my observation
p = 6
q = 4
arma_fit = arima(madness_residuals, order = c(p, 0, q))

#AIC for the model
AIC(arma_fit)
```

```
## [1] 112.4179
```

The AIC value we obtained for our initial ARMA(6,4) model is 112. This number on its own tells us little until it is compared with the AIC values from other models. The AIC is calculated based on the trade-off between the goodness of fit of the model and the complexity of the model (penalizing for the number of parameters used). The goal is to choose a model that best explains the data with the fewest parameters.

Starting with an ARMA(6,4) model based on initial analysis, we looked for the best fit for our data by testing a series of ARMA(p,q) models. We compared different p and q combinations, aiming for the lowest AIC value to get the right mix of accuracy and model simplicity.

After checking multiple models, we found ARMA(3,3) to be the best. It had the lowest AIC, showing it can capture our data's trends without being overly complex. This ARMA(3,3) model now stands as our chosen model for forecasting and analyzing the March Madness data.

```
# Print the best model's details
cat("Best ARMA model is ARMA(", best_p, ",", best_q, ") with AIC:", best_aic, "\n")
```

```
## Best ARMA model is ARMA( 3 , 3 ) with AIC: 103.1682
```

The optimal model identified for our time series analysis is the ARMA(3,3), which signifies that both the autoregressive (AR) and moving average (MA) components are of order 3. This model configuration suggests that the current value in the series is likely influenced by the preceding three values (AR part) as well as by the previous three forecast errors (MA part).

The accompanying AIC value of 103.1682 is particularly noteworthy as it is the lowest among all the models we evaluated. A lower AIC score is preferable because it indicates a model that provides a better fit to the data while using fewer parameters, thus avoiding overfitting.

```
#Chosen model is the ARMA(3,3)

p = 3
q = 3
fit = arima(madness_residuals, order = c(p, 0, q))
```

To validate our ARMA(3,3) model, we assess causality and invertibility. Causality ensures the model's current values aren't influenced by future values, while invertibility allows us to reconstruct past noise from current and past observations.

We check causality by examining the roots of the characteristic polynomial of the AR component, ensuring all roots lie outside the unit circle, indicating a diminishing influence of past values:

```
p = 3
ar_coefs = -fit$coef[1:p]
ar_roots = polyroot(c(1, ar_coefs))
ar_roots_outside_unit_circle = all(Mod(ar_roots) > 1)
cat("Causality: ", ar_roots_outside_unit_circle, "\n")
```

```
## Causality:  TRUE
```

The `polyroot` function calculates the roots, and the `Mod` function checks their magnitudes. For causality, all roots must have magnitudes greater than 1.

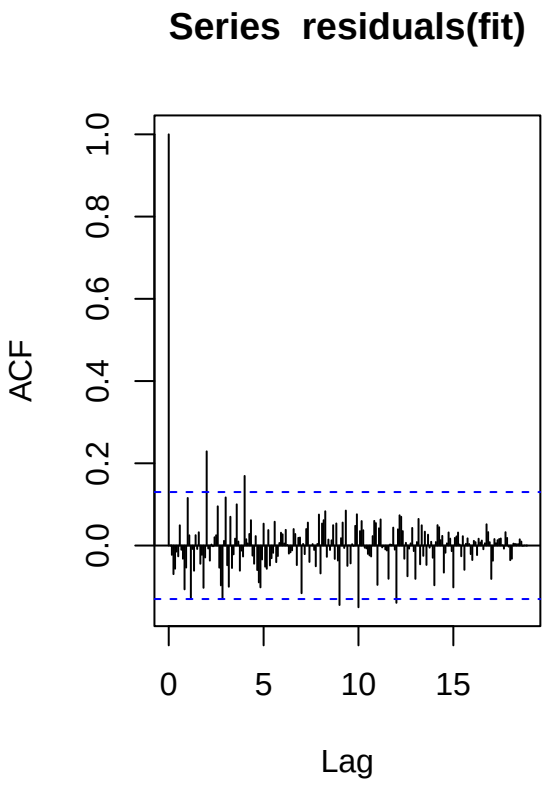
Invertibility is assessed similarly for the MA component, confirming that the model represents the process in terms of past white noise:

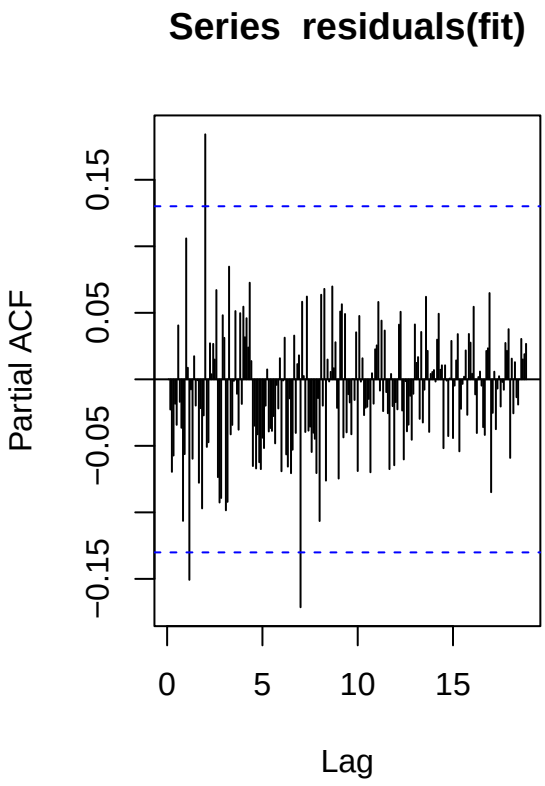
```
q = 3
ma_coefs = fit$coef[(p+1):(p+q)]
ma_roots = polyroot(c(1, ma_coefs))
ma_roots_outside_unit_circle = all(Mod(ma_roots) > 1)
cat("Invertibility: ", ma_roots_outside_unit_circle, "\n")
```

```
## Invertibility:  TRUE
```

To ensure our model sufficiently captures the data's structure, we perform residual diagnostics with ACF and PACF plots and the Ljung-Box test, which should show no significant autocorrelation:

```
par(mfrow = c(1, 2))

# Checking the residuals for any remaining autocorrelation
acf(residuals(fit), lag.max = 1000)
pacf(residuals(fit), lag.max = 1000)
```






```r
par(old_par)
```

```r
# Ljung-Box Test
Box.test(residuals(fit), lag = 10, type = "Ljung-Box")
```

```
##
##  Box-Ljung test
##
## data:  residuals(fit)
## X-squared = 5.7557, df = 10, p-value = 0.8354
```

This comprehensive assessment ensures our ARMA(3,3) model is both theoretically sound and practically reliable for analyzing the March Madness time series data.

Prediction

We're now set to make predictions for our time series, spanning a 12-month cycle from December of the current year to December of the following year. This forecast incorporates the combined effect of three key elements: the trend component, the seasonal fluctuations, and the residuals as modeled by our ARMA(3,3) model. Each element plays a crucial role in capturing the unique characteristics of the March Madness data:

1. **Trend Component**: First, we forecast the trend component by extrapolating the linear trend observed in our time series.

```
trend_coefs <- coef(lm(madness_data_trend ~ time(madness_data_trend)))
trend_forecast <- trend_coefs[1] + trend_coefs[2] * (time(madness_data_trend)[length(madness_data_trend
trend_forecast <- ts(trend_forecast, start = c(2023,12), frequency = 12)
trend_forecast
```

```
##            Jan       Feb       Mar       Apr       May       Jun       Jul
## 2023
## 2024 0.4878337 0.5125030 0.5371723 0.5618416 0.5865109 0.6111802 0.6358495
##            Aug       Sep       Oct       Nov       Dec
## 2023                                     0.4631644
## 2024 0.6605188 0.6851881 0.7098574 0.7345268
```

2. **Seasonal Component**: We use the last year's seasonal data to project the seasonal component for the next year.

```
last_year <- tail(madness_data_season, 12)
seasonal_forecast <- rep(last_year, length.out = 12)
seasonal_forecast <- ts(seasonal_forecast, start = c(2023,12), frequency = 12)
```

3. **Rough Component**: The ARMA model's predictions for the rough component, or residuals, are generated using the `forecast()` function.

```
residuals_forecast <- forecast(fit, h = 12)  # Forecasting the next year
residuals_forecast <- ts(residuals_forecast$mean, start = c(2023,12), frequency = 12)
```

Finally, we combine these forecasts to form our final prediction for the next year:

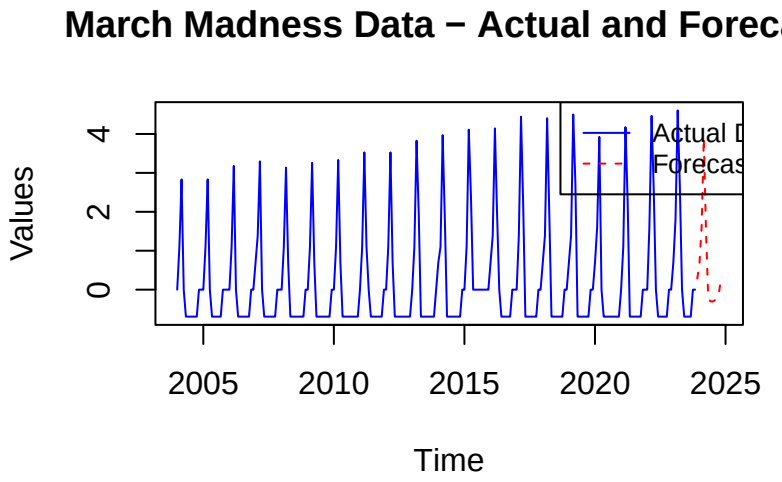
```
final_forecast <- trend_forecast + seasonal_forecast + residuals_forecast
```

This comprehensive approach integrates the trend, seasonality, and residuals, leveraging the strengths of our ARMA model and the patterns identified in the data to produce a well-rounded forecast for March Madness interest.

Final forcast

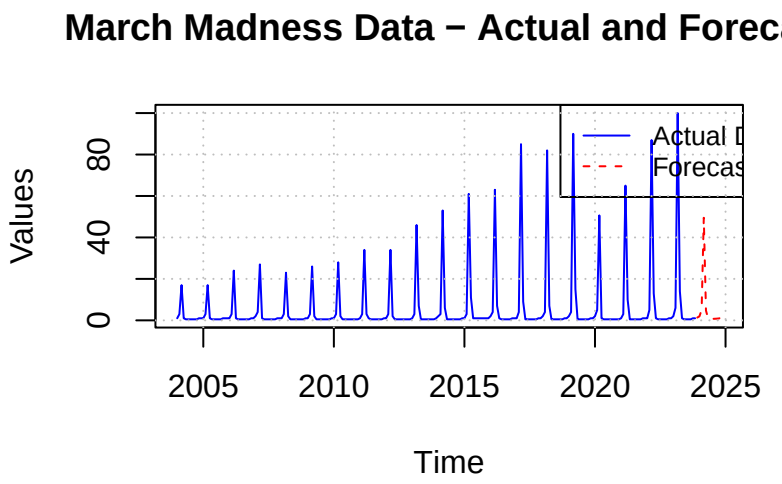
Having combined the trend, seasonal, and ARMA model forecasts, we are now ready to visualize our comprehensive forecast alongside the actual log-transformed March Madness data. This visualization will encompass data from the start of the series through to December of the following year, providing a complete picture of both historical trends and future projections.

```
par(old_par)
# Assuming madness_df_log, trend_forecast, seasonal_forecast, and residuals_forecast are ts objects

# Create a time series object for the final forecast
 # Adjust the start time to forecast from December
final_forecast_ts = ts(final_forecast, start = c(2023, 12), frequency = 12)

# Combine the actual data and the forecast for plotting
combined_ts = ts.union(madness_data_log, final_forecast_ts)
madness_data_final = exp(combined_ts)

# Plotting
plot(combined_ts, plot.type = "single", col = c("blue", "red"), lty = c(1, 2),
     xlab = "Time", ylab = "Values", main = "March Madness Data - Actual and Forecast")

# Adding a more descriptive legend outside the plot area
legend("topright", inset = c(-0.2, 0),
       legend = c("Actual Data", "Forecasted Data"),
       col = c("blue", "red"), lty = c(1, 2), cex = 0.8)
```




```
plot(madness_data_final, plot.type = "single", col = c("blue", "red"), lty = c(1, 2),
     xlab = "Time", ylab = "Values", main = "March Madness Data - Actual and Forecast")


legend("topright", inset = c(-0.2, 0),
       legend = c("Actual Data", "Forecasted Data"),
       col = c("blue", "red"), lty = c(1, 2), cex = 0.8)
```

```r
# Enhance plot with grid lines for better readability
grid(nx = NULL, ny = NULL, col = "gray", lty = "dotted", lwd = par("lwd"))
```




```r
forecast_only <- madness_data_final[time(madness_data_final) >= 2023 + 10.5/12, "final_forecast_ts"]
# Print the forecast portion
print(ts(forecast_only,start = c(2023,12),frequency = 12))
```

```
##             Jan        Feb        Mar        Apr        May        Jun
## 2023
## 2024  1.9278575  5.0190757 49.4960073  4.7293938  0.8543041  0.7415743
##             Jul        Aug        Sep        Oct        Nov        Dec
## 2023                                                          1.3295710
## 2024  0.7349887  0.7590880  0.8150824  0.9258887  1.3872337
```

Table 1: Forecasted Values for 2023-2024

Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2023	-	-	-	-	-	-	-	-	-	-	-	1.33
2024	1.93	5.02	49.50	4.73	0.85	0.74	0.73	0.76	0.82	0.93	1.39	-

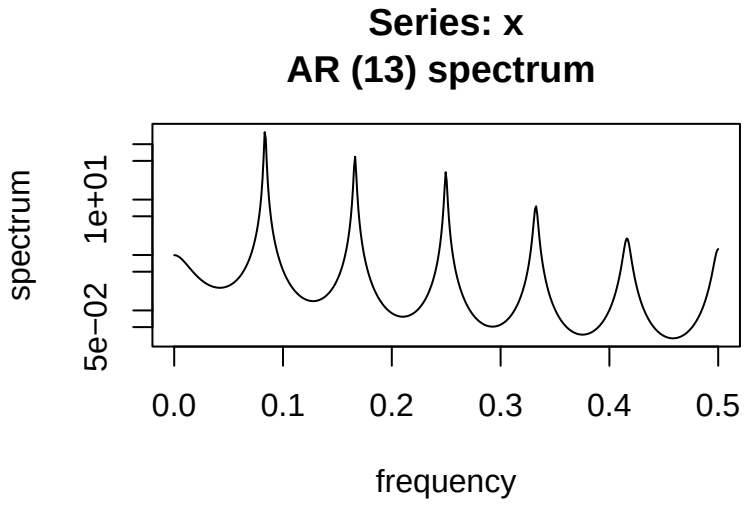
Spectral Analysis

The spectral analysis displayed in the "Power Spectrum" plot is a powerful tool for uncovering hidden periodicities in time series data. By converting the data from the time domain to the frequency domain using the Fast Fourier Transform (FFT), we can observe the distribution of power across various frequency components.

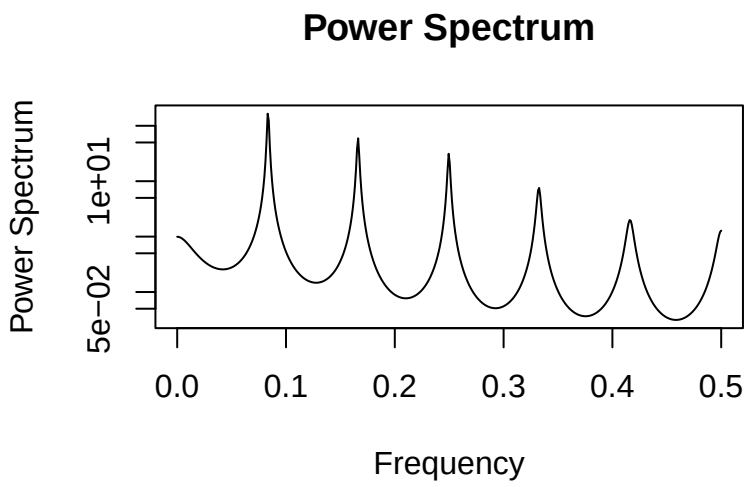
In this analysis:

- We first convert our `madness_df_log$Madness` time series data into a numeric vector, `madness_numeric`.
- We then apply the FFT to this numeric vector to obtain the frequency components of the data.
- The power spectrum is calculated as the squared modulus of the FFT result, which represents the contribution of each frequency to the overall variability in the data.
- The plot generated by the `spectrum()` function using an autoregressive (AR) method displays the power spectrum as a function of frequency.

```
madness_numeric <- as.numeric(madness_df_log$Madness)
fft_result <- fft(madness_numeric)
power_spectrum <- Mod(fft_result)^2
spec <- spectrum(madness_numeric, method = "ar")
```




```
plot(spec, main = "Power Spectrum", xlab = "Frequency", ylab = "Power Spectrum")
```



The peaks in the "Power Spectrum" plot indicate the dominant frequencies or cycles in the data. These peaks are crucial for understanding the underlying seasonal or cyclical behavior in the March Madness interest over time. Identifying these frequencies allows us to model the time series data more accurately by incorporating seasonal ARIMA (SARIMA) models, harmonic regression, or other techniques that account for periodic behavior.

Spectral analysis can guide the selection of model parameters, such as the order of differencing in seasonal decomposition, or identify hidden periodicities that may not be captured by traditional time series decomposition methods. By leveraging this analysis, we aim to enhance the accuracy of our predictions and gain a deeper understanding of the temporal dynamics at play.

Discussion

Our analysis of the March Madness data revealed distinct seasonal patterns, particularly a prominent spike in interest each March. This aligns with the event's timing and is a key feature our model needed to capture. While we successfully identified the ARMA(3,3) model as the best fit based on the lowest AIC value, there were challenges and limitations along the way.

One notable difficulty was in forecasting. We resorted to using the forecast function to extend our predictions 12 months ahead. This choice, while practical, might have limited our control over the forecasting process and the specific nuances of the model's behavior.

Another challenge was the need to estimate certain data points, like for March 2020 due to COVID-19 disruptions and for values less than 1, which we set to 0.5. These estimations, though necessary, introduced an element of uncertainty and potential bias in our model.

The ACF and PACF plots did not provide as clear guidance as we had hoped, leading us to rely heavily on AIC for model selection. There was also a slight residual autocorrelation after fitting the ARIMA model, which we couldn't entirely eliminate. This residual autocorrelation suggests that there might be aspects of the data not fully captured by the model.

Despite these challenges, the model fits well overall, with the forecasted values appearing realistic and in line with expected patterns. The significant peak in March and the lower interest in other months strongly reflect the real-world interest in March Madness.

Conclusion

Our time series analysis of March Madness data effectively captures the cyclical nature of public interest in the event. The forecast for January 2024 to December 2024 shows a pronounced peak in March, consistent with when the tournament occurs, followed by a normalization of interest. This pattern demonstrates the effectiveness of our model in reflecting the event's annual popularity cycle.

While there were challenges, such as data estimation and slight residual autocorrelation, the ARMA(3,3) model proved to be a robust choice, balancing complexity with fit. The insights provided by this analysis can be valuable to stakeholders in marketing, event planning, and sports analysis, offering a predictive view of public engagement with March Madness.

This analysis underscores the importance of critical evaluation in statistical modeling. Acknowledging and addressing the limitations of our approach not only adds credibility to our findings but also opens avenues for future improvement and more nuanced analyses.

References

- Cowpertwait, P.S.P., & Metcalfe, A.V. (2009). *Introductory Time Series with R.* Springer.
- Shumway, R.H., & Stoffer, D.S. (2011). *Time Series Analysis and Its Applications: With R Examples* (3rd ed.). Springer.
- Brockwell, P.J., & Davis, R.A. (2002). *Introduction to Time Series and Forecasting* (2nd ed.). Springer.

Appendix

```
knitr::opts_chunk$set(echo = TRUE)

library(forecast)
library(lubridate)
library(tidyverse)
library(ggplot2)
library(dplyr)
library(tseries)
library(grid)

# Load the dataset from a CSV file
data = read.csv('march_madness_data.csv')

# Rename the columns for clarity
colnames(data) = c('month', 'march_madness')

#Sample output of the data
head(data,6)

# Convert '<1' values to 0.5
data$march_madness = as.numeric(gsub("<1", "0.5", data$march_madness))

# Calculate the average for March in previous years
march_rows = data %>% filter(grepl("03$", month) & substr(month, 1, 4) != "2020")
average_march_madness = mean(march_rows$march_madness, na.rm = TRUE)

# Impute missing value for March 2020 with the calculated average
data$march_madness[data$month == "2020-03"] = average_march_madness

# Convert the 'month' column to a year-month format and create a time series object
data$month = ym(data$month)

#Time series object
madness_data = ts(data$march_madness, start = 2004, frequency = 12)

# Create a DataFrame for visualization purposes
madness_df = data.frame(
  Time = time(madness_data),
  Madness = as.numeric(madness_data)
)


# Plot the time series
plot(madness_data, type='l', main='March Madness Data', xlab='Year', ylab='Interest')

#Applying log transformation since the data doesnt have constant variance
madness_data_log = log(madness_data)

madness_df_log = data.frame(
  Time = time(madness_data_log),
  Madness = as.numeric(madness_data_log)
)
```

```r
#Log Transformed data plot
plot(madness_data_log, type='l', main='March Madness Data', xlab='Year', ylab='Interest')


calculate_trend <- function(timeseries_data, d) {
  # Apply the moving average filter
  ma <- stats::filter(timeseries_data, sides = 2, rep(1, d + 1)/(d + 1))

  # Calculate the number of leading NAs introduced by the filter
  na_count <- ifelse(d %% 2 == 0, d/2, (d + 1)/2)

  # Create the time series object starting after the NAs
  ts_start <- start(timeseries_data) + c(0, na_count/2)

  # Return the time series without the leading and trailing NAs
  return(ts(ma, start = ts_start, frequency = frequency(timeseries_data)))
}

#Choosing d to be 12 since the period is 12 months, on a yearly basis
d = 12

#Applying trend calculation function
madness_data_trend = calculate_trend(madness_data_log,d)


# Plot the original data
plot(madness_data_log, type = 'l', main = "March Madness Data with Moving Average")
# Overlay the moving average, skipping NA values
lines(madness_data_trend, col = "red")
# Trim the NA values from the moving average to find the detrended  data
madness_data_trend = na.omit(madness_data_trend)

# Corresponding indices of the non-NA values in the moving average
valid_indices = which(!is.na(madness_data_trend))

# Trim the original time series to the same length as the moving average
madness_data_log_trimmed = ts(madness_data_log[valid_indices],start = start(madness_data_trend),frequen

# Compute the detrended data
detrended_data = madness_data_log_trimmed - madness_data_trend

#plotting the detrended data
plot(detrended_data,type = 'l', main = "March Madness Detrended Data")


#Defining the seasonality estimation function
calculate_seasonality = function(detrended_data, d) {
  n = length(detrended_data)
  num_years = floor(n / d)
  seasonal_effect = rep(NA, d)

  for (k in 1:d) {
```

```
    seasonal_indices = seq(k, n, by = d)
    seasonal_effect[k] = mean(detrended_data[seasonal_indices], na.rm = TRUE)
  }
  seasonal_effect = seasonal_effect - mean(seasonal_effect, na.rm = TRUE)

  return(ts(rep(seasonal_effect, length.out = n),start = start(detrended_data), frequency = 12))
}
#Using function to calculate seasonality
madness_data_season = calculate_seasonality(detrended_data,d)

#Plotting the data
plot(madness_data_season, type = 'l',
     main = "Seasonality Component of March Madness",
     xlab = "Time",
     ylab = "Seasonal Effect")



# Adjusting layout for a 2x2 grid
par(mfrow = c(2, 2), mar = c(4, 4, 2, 1))

# Plot 1: Original data with date
plot(madness_data_log, type = 'l', main = "March Madness Interest Over Time",
     xlab = "Time", ylab = "Log of Interest")

# Plot 2: Data without trend
x_trend = madness_data_log_trimmed - madness_data_trend
plot(x_trend, type = 'l', main = "March Madness Data without Trend",
     xlab = "Time", ylab = "Detrended Data")

# Plot 3: Data without seasonality
x_season = madness_data_log_trimmed - madness_data_season
plot(x_season, type = 'l', main = "March Madness Data without Season",
     xlab = "Time", ylab = "Deseasonalized Data")

# Plot 4: Original data with trend and seasonality lines
plot(madness_data_log_trimmed, type = 'l', main = "March Madness Interest",
     xlab = "Index", ylab = "Log of Interest")
lines(madness_data_trend, col = 'red')
lines(madness_data_season, lty = 4, col = 'green')


plot_par <- par(no.readonly = TRUE)

# Adjust margins and layout for three stacked plots
par(mfrow = c(3, 1), mar = c(4, 4, 2, 1), oma = c(0, 0, 2, 0))

# Set common axis limits
xlims <- range(time(madness_data_log_trimmed))
ylims <- range(madness_data_log_trimmed, madness_data_trend, madness_data_season, na.rm = TRUE)

# Plot 1: Original data
plot(madness_data_log_trimmed, type = 'l', main = "March Madness Interest",
```

```
     xlab = "Index", ylab = "Log of Interest", xlim = xlims, ylim = ylims, col = 'black')

# Plot 2: Trend Component
plot(madness_data_trend, type = 'l', main = "Trend Component",
     xlab = "Index", ylab = "Trend", xlim = xlims, ylim = ylims, col = 'red')

# Plot 3: Seasonal Component
plot(madness_data_season, type = 'l', main = "Seasonal Component",
     xlab = "Index", ylab = "Seasonality", xlim = xlims, ylim = ylims, col = 'green')

# Restore original par settings
par(plot_par)

# Plot Data without trend and seasonality (residuals)
madness_residuals = madness_data_log_trimmed - madness_data_season - madness_data_trend
#Finding the mean of the residuals
x_bar_residuals = mean(madness_residuals)
#Residuals Plot
plot(madness_residuals, type = 'l', main = "March Madness Residuals", xlab = "Time", ylab = "Residuals"


# Adding a dashed red line at the mean of the residuals
abline(h = x_bar_residuals, col = "red", lty = "dashed")

old_par <- par(no.readonly = TRUE)

#Setting up display layout
par(mfrow = c(1, 2))

# ACF and PACF Plots
acf(madness_residuals,lag = 50, main = "ACF of Residuals", xlab = "Lag", ylab = "ACF")
pacf(madness_residuals,lag = 50, main = "PACF of Residuals", xlab = "Lag", ylab = "PACF")

par(old_par)
# QQ Plot for Normality
qqnorm(madness_residuals)
qqline(madness_residuals, col = "red")
#Heavy tails, not too good
# Normality Test
shapiro.test(madness_residuals)
lambda = BoxCox.lambda(madness_residuals)
transformed_residuals = BoxCox(madness_residuals, lambda)
par(mfrow = c(1, 2))

#recheck ACF and PACF
acf(transformed_residuals, lag.max = 50, main = "ACF of Transformed Residuals", xlab = "Lag", ylab = "A
pacf(transformed_residuals, lag.max = 50, main = "PACF of Transformed Residuals", xlab = "Lag", ylab =


par(mfrow = c(1, 2))

# Reassess normality with QQ plot and Shapiro-Wilk test
qqnorm(transformed_residuals)
```

```r
qqline(transformed_residuals, col = "red")

# Plot the transformed residuals
plot(transformed_residuals, type = 'l',
     main = "Transformed Residuals",
     xlab = "Time Index",
     ylab = "Transformed Residual Values")#Much better qq

par(old_par)
shapiro.test(transformed_residuals)

#Chosen Model from my observation
p = 6
q = 4
arma_fit = arima(madness_residuals, order = c(p, 0, q))

#AIC for the model
AIC(arma_fit)

#Setting maximum for p and q terms to 5 for simplicity in the model
max_p = 5
max_q = 5

# Initialize variables to store the best model's information
best_aic = Inf
best_p = NA
best_q = NA
best_model = NULL

# Nested loops to iterate over p and q values
for (p in 0:max_p) {
    for (q in 0:max_q) {
        # Fit ARMA model
        model = arima(madness_residuals, order = c(p, 0, q))

        # Calculate AIC
        aic_value = AIC(model)

        # Check if this model has the best (lowest) AIC so far
        if (aic_value < best_aic) {
            best_aic = aic_value
            best_p = p
            best_q = q
            best_model = model
        }
    }
}
# Print the best model's details
cat("Best ARMA model is ARMA(", best_p, ",", best_q, ") with AIC:", best_aic, "\n")

#Chosen model is the ARMA(3,3)

p = 3
```

```r
q = 3
fit = arima(madness_residuals, order = c(p, 0, q))
p = 3
ar_coefs = -fit$coef[1:p]
ar_roots = polyroot(c(1, ar_coefs))
ar_roots_outside_unit_circle = all(Mod(ar_roots) > 1)
cat("Causality: ", ar_roots_outside_unit_circle, "\n")
q = 3
ma_coefs = fit$coef[(p+1):(p+q)]
ma_roots = polyroot(c(1, ma_coefs))
ma_roots_outside_unit_circle = all(Mod(ma_roots) > 1)
cat("Invertibility: ", ma_roots_outside_unit_circle, "\n")
par(mfrow = c(1, 2))

# Checking the residuals for any remaining autocorrelation
acf(residuals(fit), lag.max = 1000)
pacf(residuals(fit), lag.max = 1000)

par(old_par)
# Ljung-Box Test
Box.test(residuals(fit), lag = 10, type = "Ljung-Box")

trend_coefs <- coef(lm(madness_data_trend ~ time(madness_data_trend)))
trend_forecast <- trend_coefs[1] + trend_coefs[2] * (time(madness_data_trend)[length(madness_data_trend)
trend_forecast <- ts(trend_forecast, start = c(2023,12), frequency = 12)
trend_forecast
last_year <- tail(madness_data_season, 12)
seasonal_forecast <- rep(last_year, length.out = 12)
seasonal_forecast <- ts(seasonal_forecast, start = c(2023,12), frequency = 12)
residuals_forecast <- forecast(fit, h = 12)  # Forecasting the next year
residuals_forecast <- ts(residuals_forecast$mean, start = c(2023,12), frequency = 12)
final_forecast <- trend_forecast + seasonal_forecast + residuals_forecast
par(old_par)
# Assuming madness_df_log, trend_forecast, seasonal_forecast, and residuals_forecast are ts objects

# Create a time series object for the final forecast
 # Adjust the start time to forecast from December
final_forecast_ts = ts(final_forecast, start = c(2023, 12), frequency = 12)

# Combine the actual data and the forecast for plotting
combined_ts = ts.union(madness_data_log, final_forecast_ts)
madness_data_final = exp(combined_ts)

# Plotting
plot(combined_ts, plot.type = "single", col = c("blue", "red"), lty = c(1, 2),
     xlab = "Time", ylab = "Values", main = "March Madness Data - Actual and Forecast")

# Adding a more descriptive legend outside the plot area
legend("topright", inset = c(-0.2, 0),
       legend = c("Actual Data", "Forecasted Data"),
       col = c("blue", "red"), lty = c(1, 2), cex = 0.8)
```

```r
plot(madness_data_final, plot.type = "single", col = c("blue", "red"), lty = c(1, 2),
     xlab = "Time", ylab = "Values", main = "March Madness Data - Actual and Forecast")


legend("topright", inset = c(-0.2, 0),
       legend = c("Actual Data", "Forecasted Data"),
       col = c("blue", "red"), lty = c(1, 2), cex = 0.8)

# Enhance plot with grid lines for better readability
grid(nx = NULL, ny = NULL, col = "gray", lty = "dotted", lwd = par("lwd"))


forecast_only <- madness_data_final[time(madness_data_final) >= 2023 + 10.5/12, "final_forecast_ts"]
# Print the forecast portion
print(ts(forecast_only,start = c(2023,12),frequency = 12))


madness_numeric <- as.numeric(madness_df_log$Madness)
fft_result <- fft(madness_numeric)
power_spectrum <- Mod(fft_result)^2
spec <- spectrum(madness_numeric, method = "ar")
plot(spec, main = "Power Spectrum", xlab = "Frequency", ylab = "Power Spectrum")
```